АЛГОРИТМЫ

Вводный курс

ALGORITHMS UNLOCKED

Thomas H. Cormen



The MIT Press
Cambridge, Massachusetts London, England

АЛГОРИТМЫ

Вводный курс

Томас Х. Кормен



"Вильямс"
Москва • Санкт-Петербург • Киев
2014

ББК 32.973.26-018.2.75
К66
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция канд. техн. наук *И.В. Красикова*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Кормен, Томас Х.

К66 Алгоритмы: вводный курс. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2014. — 208 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1868-0 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Томас Х. Кормен

Алгоритмы: вводный курс

Литературный редактор *Л.Н. Красножон*
Верстка *М.А. Удалов*
Художественный редактор *Е.П. Дынник*
Корректор *Л.А. Гордиенко*

Подписано в печать 11.11.2013. Формат 70x100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 16,7. Уч.-изд. л. 18,8.
Тираж 1500 экз. Заказ № 3919.

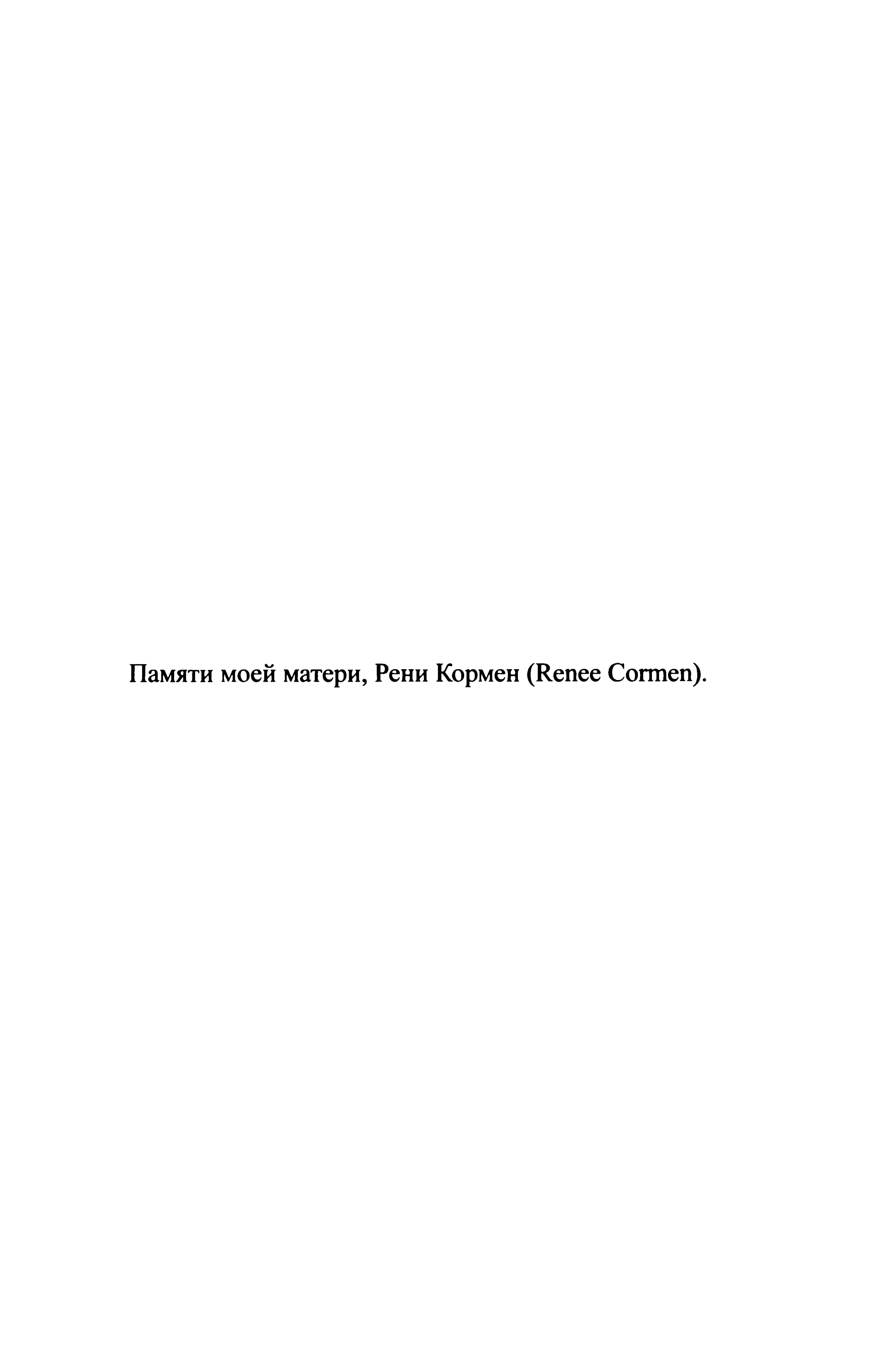
Первая Академическая типография "Наука"
199034, Санкт-Петербург, 9-я линия, 12/28

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1868-0 (рус.)
ISBN 978-0-262-51880-2 (англ.)

Оглавление

Содержание



Памяти моей матери, Рени Кормен (Renee Cormen).

Предисловие

Как компьютеры решают задачи? Как ваш маленький GPS в считанные секунды находит самый быстрый путь из несметного множества возможных маршрутов? Когда вы покупаете что-то в Интернете, как обеспечивается защита номера вашей кредитной карты от перехвата злоумышленником? Ответом на эти и массу других вопросов являются ***алгоритмы***. Я написал эту книгу, чтобы раскрыть вам тайны алгоритмов.

Я — соавтор учебника *Алгоритмы: построение и анализ*. Это замечательная книга (конечно, я небеспристрастен), но местами она представляет собой практически научный труд.

Книга, которую вы держите в своих руках, — совершенно иная. Это даже не учебник. Она не погружается в алгоритмы достаточно глубоко, не охватывает их разнообразие сколь-нибудь широко, не учит методам проектирования компьютерных алгоритмов, и в ней даже нет задач и упражнений, которые должен решать читатель! Так что же представляет собой эта книга? Это отправная точка для вас, если вы

- интересуетесь тем, как компьютеры решают поставленные перед ними задачи;
- хотите знать, как оценить качество этих решений;
- хотите понимать, как задачи, решаемые компьютерами, и используемые для этого методы связаны с реальным, некомпьютерным миром;
- не очень сильны в математике;
- не написали ни одной программы (впрочем, умение программировать нисколько не мешает чтению данной книги, даже наоборот).

Некоторые книги о компьютерных алгоритмах концептуальны, с небольшим количеством технических деталей. Некоторые из них переполнены технически точными описаниями. Ряд книг находится между этими крайностями. Для каждого типа книг есть свое место и свой читатель. Я бы поместил эту книгу в промежуточную категорию. Да, в ней есть немного математики, и иногда она довольно глубоко погружается в детали, но я старался избегать таких мест (за исключением, возможно, конца книги, где я уже просто не мог контролировать себя).

Я представляю эту книгу своеобразной закуской. Представьте, что вы зашли в ресторан и для начала заказали закуски, решив подождать с основным заказом до тех пор, пока не справитесь с этой мелочью. Ваш заказ принесен, вы пробуете его. Возможно, еда вам не понравится, и вы решите уйти из этого ресторана. Возможно, вы утолите голод одними салатиками. А может быть, вам так понравится, что вы закажете официанту обильный обед и с нетерпением будете его ждать. Рассматривая эту книгу как закуску, я надеюсь, что либо вы полностью насытитесь ею и сочтете, что достаточно погрузились в мир алгоритмов, либо прочитанное заинтересует вас настолько, что вы захотите узнать побольше. Каждая глава заканчивается разделом “Дальнейшее чтение”, который подскажет вам, что прочесть для углубленного понимания вопросов.

Чему научит вас эта книга

Я не знаю, чему научит вас эта книга. Я могу только сказать, что именно я постарался вложить в эту книгу, надеясь, что после ее прочтения вы будете знать следующее.

- Что такое компьютерные алгоритмы, как их описать и оценить.
- Простые способы поиска информации в компьютере.
- Методы переупорядочения информации в компьютере некоторым предопределенным способом (мы называем эту задачу "сортировка").
- Как решаются базовые задачи, которые можно смоделировать в компьютере с помощью математической структуры, известной как "граф". Среди множества приложений графы прекрасно подходят для моделирования дорожных сетей (между какими перекрестками есть непосредственно связывающие их дороги и какой они длины?), взаимосвязей между заданиями (какое задание должно предшествовать другим?), финансовых отношений (каковы курсы обмена между разными валютами?) или взаимоотношений между людьми (кто с кем знаком? кто кого ненавидит? какой актер снимался в фильме с некоторым другим актером?).
- Как решаются задачи, в которых участвуют строки текстовых символов. Некоторые из этих задач находят применение в таких областях, как биология, где символы представляют собой базовые аминокислоты, а строки символов — структуры ДНК.
- Основные принципы, лежащие в основе криптографии. Даже если вы никогда не шифровали сообщений сами, ваш компьютер, вероятно, не раз это делал, например при покупке товаров через Интернет.
- Фундаментальные идеи сжатия данных, выходящие далеко за рамки сокращений, например, в столь любимых недалекой молодежью смсках.
- Что некоторые задачи слишком трудны, чтобы решить их на компьютере за любое разумное время (или как минимум никто пока что не нашел способа их решения за приемлемое время).

Что следует знать для понимания материала книги

Как я говорил ранее, в книге есть немного математики. Если это пугает вас до дрожи в коленках, можете попробовать пропускать ее или поискать менее техническую книгу. Но я сделал все возможное, чтобы сделать те крохи математики, которые есть в книге, доступными для всех.

Я не думаю, что вы никогда не писали и не читали ни одной компьютерной программы. Если вы в состоянии следовать инструкциям, написанным обычным языком, то должны быть в состоянии понять, как я выражаю в книге составляющие алгоритм шаги. Если вы засмеетесь над следующей шуткой, вы на верном пути.

Вы слышали о программисте, который застрял в душе? Он[1] *мыл голову и строго следовал инструкции на бутылке шампуня, в которой было написано "Намылить. Вспенить. Прополоскать. Повторить."*[2]

В книге я использовал довольно неформальный стиль написания, надеясь, что индивидуальный подход поможет сделать материал более доступным. Некоторые главы зависят от материала предыдущих глав, но такая зависимость характерна только для некоторых из них. Ряд глав начинается совершенно не технически, но постепенно принимает все более технический характер. Если вы обнаружите, что материал одной главы спокойно укладывается в вашей голове, значит, очень велики шансы на то, что вы поймете по крайней мере начало следующей главы.

Если вы нашли ошибку

Если вы нашли ошибку в книге, сообщите мне о ней электронной почтой по адресу `unlocked@mit.edu`.

Благодарности

Большая часть материала данной книги взята из книги *Алгоритмы: построение и анализ*, так что моя первая и самая горячая благодарность — моим соавторам по этой книге Чарльзу Лейзерсону (Charles Leiserson), Рону Ривесту (Ron Rivest) и Клиффу Штайну (Cliff Stein). Я так часто ссылаюсь на эту книгу[3], что использую в тексте сокращение CLRS — по первым буквам фамилий авторов. Работа над книгой, которую вы сейчас держите в руках, ясно показала мне, как мне недостает сотрудничества с Чарльзом, Роном и Клиффом. Транзитивно я также благодарю всех, кого мы благодарили в предисловии к CLRS.

Я также воспользовался материалом, который преподавал в Дартмуте, в особенности на курсах информатики 1, 5 и 25. Я благодарен своим студентам, вопросы которых помогли мне выработать педагогический подход, понять, что им интересно, и по каменному молчанию вычислить не заинтересовавшие их темы.

Эта книга написана по предложению Ады Бранштейн (Ada Brunstein), нашего редактора в MIT Press при подготовке третьего издания CLRS. В настоящее время ее место занимает Джим ДеВольф (Jim DeWolf). Первоначально книга задумывалась как одна из книг серии MIT Press "Essential Knowledge" ("Базовые знания"), но, как оказалось, MIT Press — слишком научное издательство для выпуска такой серии. Джим справился с этой неловкой ситуацией, позволив мне написать то, что хотел бы написать я сам, а не то, что задумы-

---

[1] *Или она. Политкорректность заставляет написать "она", но соотношение полов в этой области деятельности (и опасения, что книгу будут читать феминистки) заставляет написать "он".*

[2] *Бывают ситуации и похуже — один такой программист так и не смог помыть голову, потому что на бутылке была надпись "для сухих волос", а всухую шампунь никак не хотел пениться...* — Примеч. пер.

[3] *Имеется ее перевод на русский язык: Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. Алгоритмы: построение и анализ, 3-е изд. — М.: Издательский дом "Вильямс", 2013.*

валось MIT Press первоначально. Я также высоко ценю поддержку Эллен Фаран (Ellen Faran) и Гита Деви Манактала (Gita Devi Manaktala) из MIT Press.

Техническим редактором 2- и 3-го изданий CLRS была Джули Суссман (Julie Sussman), и я очень хотел, чтобы она взялась и за эту книгу. Это наилучший технический редактор, обладающий к тому же наилучшим чувством юмора. Посмотрите сами, какое письмо, посвященное черновому варианту главы 5, прислала мне Джули.

Власти объявили о розыске сбежавшей главы, которая, как оказалось, скрывается в вашей книге. Мы не можем выяснить, из какой книги она совершила побег, но и не можем представить себе, как она могла бы прятаться в вашей книге многие месяцы без вашего ведома. Поэтому у нас нет выбора, кроме как привлечь вас к ответственности. Надеемся, что вы возьмете на себя задачу перевоспитания данной главы, что даст ей возможность стать продуктивным гражданином вашей книги. Доклад сотрудника, произведшего арест, — Джули Суссман — прилагается.

Гугол спасибо тебе, Джули!

По профессии я далек от криптографии, и в главе о принципах криптографии использовалось множество комментариев и предложений Рона Ривеста (Ron Rivest), Шона Смита (Sean Smith), Рейчел Миллер (Rachel Miller) и Хиджи Рэйчел Линь (Huijia Rachel Lin). В этой главе имеется примечание о бейсбольных знаках, и я благодарю Боба Уолена (Bob Whalen), тренера по бейсболу в Дартмуте, за терпеливое пояснение мне некоторых систем знаков в бейсболе. Илана Арбиссер (Ilana Arbisser) проверила, что вычислительная биология выравнивает последовательности ДНК именно так, как я пояснил в главе 7, “Алгоритмы на строках”. Мы с Джимом ДеВольфом (Jim DeWolf) перепробовали множество названий для нашей книги, но окончательный вариант был предложен дартмутским студентом Чандером Рамешем (Chander Ramesh).

Факультет информатики колледжа в Дартмуте — удивительное место работы! Мои коллеги составляют настолько блестящий профессиональный коллектив, что равного ему не найти. Если вы ищете место для учебы в этой области, я всерьез предлагаю вам подумать о Дартмуте.

Наконец я выражаю благодарности своей жене Николь (Nicole), а также мои родителям Рени (Renee) и Перри (Perry) Корменам (Cormen) и родителям Николь — Колетт (Colett) и Полю (Paul) Сейдж (Sage) за их любовь и поддержку. Кстати, мой отец уверен, что рисунок на с. 16 изображает цифру 5, а не букву S.

Том Кормен

Ганновер, Нью-Гэмпшир
Ноябрь 2012

1...Что такое алгоритмы и зачем они нужны

Начну с часто задаваемого вопроса “Что такое алгоритм?”

Обобщенный ответ — “набор шагов для выполнения задачи”. У вас есть алгоритмы, которые выполняются в вашей повседневной жизни. Например, алгоритм чистки зубов: открыть тюбик зубной пасты, взять зубную щетку, выдавливать зубную пасту на щетку до тех пор, пока не будет достаточно для покрытия щетки, закрыть тюбик, поместить щетку в один из квадрантов вашего рта, перемещать щетку вверх и вниз N секунд и т.д. Если вам приходится ездить на работу, у вас, конечно же, имеется алгоритм выбора транспорта и мест пересадки. У вас наверняка имеются и многие другие алгоритмы…

Однако данная книга посвящена алгоритмам, выполняемым на компьютерах, или, более обобщенно, на вычислительных устройствах. Эти алгоритмы так же влияют на вашу повседневную жизнь, как и алгоритмы, которые выполняете *вы сами*. Вы используете GPS для поиска маршрута поездки? Работает алгоритм поиска кратчайшего пути. Покупаете что-то в Интернете? Значит, вы используете (или по крайней мере должны использовать) защищенный веб-сайт, на котором работает алгоритм шифрования. Когда вы делаете покупки в Интернете, они доставляются службой доставки? Она использует алгоритм распределения заказов по машинам, а затем определяет порядок, в котором каждый водитель должен доставить пакеты. Алгоритмы работают на компьютерах везде — на вашем ноутбуке, на серверах, на вашем смартфоне, во встроенных системах (таких, как автомобиль, микроволновая печь или системы климат-контроля) — абсолютно везде!

В чем же отличие алгоритма, который работает на компьютере, от алгоритма, который выполняете вы сами? Вы можете стерпеть, если алгоритм описан неточно, но вот компьютер не столь терпелив. Например, если вы едете на работу, работающий у вас в голове алгоритм может сказать “если дорога перегружена, выбирай другой маршрут”. Вы можете понять, что значит перегруженная дорога, но компьютеру такие тонкости неизвестны…

Таким образом, компьютерный алгоритм представляет собой набор шагов для выполнения задачи, описанных достаточно точно для того, чтобы компьютер мог их выполнить. Если вам приходилось немного программировать на Java, C, C++, Python, Fortran, Matlab или тому подобных языках программирования, то у вас есть некоторое представление о требуемом уровне точности описания. Если же вы никогда не писали компьютерных программ, то, возможно, вы прочувствуете этот уровень точности, глядя на то, как я описываю алгоритмы в этой книге.

Давайте перейдем к следующему вопросу: “Чего мы хотим от компьютерного алгоритма?” Компьютерные алгоритмы решают вычислительные задачи. С учетом этого от компьютерного алгоритма требуются две вещи: он должен всегда давать правильное решение поставленной задачи и при этом эффективно использовать вычислительные ресурсы. Рассмотрим по очереди оба эти пожелания.

Корректность

Что это означает — "получение правильного решения задачи"? Обычно мы можем точно определить, что повлечет за собой правильное решение. Например, если ваш GPS выдает правильное решение задачи поиска наилучшего маршрута для путешествия, то это будет маршрут, по которому вы доберетесь в желаемый пункт назначения быстрее, чем при поездке по любому другому маршруту. Возможен поиск маршрута, который имеет наименьшую возможную длину, или маршрута, который позволит добраться побыстрее и при этом избежать дорожных сборов. Конечно, информация, которую ваш GPS использует для определения маршрута, может не соответствовать действительности. Так, с одной стороны, можно считать, что время прохождения маршрута равно расстоянию, деленному на максимально разрешенную скорость на данной дороге. Но когда дорога перегружена, а вам надо добраться побыстрее, GPS может дать вам плохой совет. Таким образом, алгоритм может быть правильным даже при неверных входных данных. Ведь если бы в описанном случае GPS получил корректные входные данные, предложенный им маршрут был бы и в самом деле самым быстрым.

Для некоторых задач достаточно сложно — а то и попросту невозможно — сказать, дает ли алгоритм верное решение задачи. Например, в случае оптического распознавания показанного на рисунке символа из 11×6 точек какой ответ считать верным — 5 или S?



Одни люди видят здесь цифру 5, в то время как другие уверенно утверждают, что это буква S. И как после этого решить, корректно или некорректно то или иное решение компьютера? К счастью, в этой книге мы ограничимся компьютерными алгоритмами, которые имеют распознаваемые решения.

Бывает, однако, что можно считать решением алгоритм, который иногда дает некорректные результаты, если только мы в состоянии контролировать, как часто это происходит. Хорошим примером является шифрование. Распространенная криптосистема RSA основана на определении того, что большое число — действительно большое, сотни десятичных знаков — является простым. Если вы писали программы, то, наверное, сумеете написать и программу проверки, является ли некоторое число n простым. Такая программа может проверять все числа от 2 до $n-1$, и если хоть одно из этих чисел является делителем n, то n — составное число. Если между 2 и $n-1$ нет ни одного делителя n, то число n — простое. Но если n — длиной в несколько сотен десятичных знаков, то кандидатов в делители гораздо, гораздо больше, чем самый быстрый компьютер в состоянии проверить за реальное время. Конечно, можно выполнить определенную оптимизацию, например не проверять четных кандидатов или ограничиться кандидатами, меньшими, чем $\sqrt{n}$ (поскольку, если d больше $\sqrt{n}$ и d является делителем n, то n/d меньше $\sqrt{n}$ и также является делителем n; следовательно, если n имеет делитель, то он будет найден до достижения проверкой $\sqrt{n}$). Если n имеет сотни десятичных знаков, то хотя $\sqrt{n}$ и имеет всего лишь около половины десятичных знаков n, это все равно очень большое число. Хорошая но-

вость заключается в том, что имеется алгоритм, который в состоянии быстро определить, является ли заданное число простым. Плохая новость в том, что этот алгоритм может делать ошибки. Конкретно — если алгоритм говорит, что некоторое число составное, то оно, определенно, составное. Но если он говорит, что число простое, то есть небольшой шанс, что это число на самом деле составное. Но плохая новость плоха не совсем: мы можем управлять частотой ошибок алгоритма, делая их действительно редкими, например одна ошибка из 2^{50} случаев. Это достаточно редко — одна ошибка на миллион миллиардов случаев, — чтобы данный алгоритм можно было применять в RSA для определения, простое ли данное число.

Корректность — достаточно сложный вопрос в случае другого класса алгоритмов, называемого приближенными алгоритмами. Приближенные алгоритмы применяются к задачам оптимизации, в которых мы хотим найти наилучшее решение в смысле некоторой количественной меры. Одним из примеров является поиск наиболее быстрого маршрута, для которого количественная мера представляет собой время поездки. Для некоторых задач нет алгоритма, который мог бы найти оптимальное решение за любое разумное количество времени, но при этом известен приближенный алгоритм, который за разумное количество времени может найти решение, которое является почти оптимальным. "Почти оптимальный" обычно означает, что отличие количественной меры найденного алгоритмом приближения от таковой для оптимального решения находится в некоторых известных пределах. Если мы в состоянии указать эти пределы отличия от оптимального решения, то можем говорить о том, что корректным решением приближенного алгоритма является любое решение, которое находится в указанных пределах.

Использование ресурсов

Что означает, что алгоритм эффективно использует вычислительные ресурсы? Об одном из показателей эффективности мы уже упомянули при обсуждении приближенных алгоритмов — это время работы алгоритма. Алгоритм, который дает правильное решение, но требует большого времени для его получения, не имеет практической ценности. Если бы вашему GPS требовался час, чтобы определить маршрут движения, стали бы вы тратить силы, чтобы его включить? Время является основным показателем эффективности, который мы используем для оценки алгоритма — конечно, после того, как мы показали, что алгоритм дает правильное решение. Но это не единственная мера эффективности. Следует также учитывать, какое количество памяти компьютера требуется алгоритму для работы, так как иначе может оказаться, что он просто не станет работать из-за недостатка памяти в вашем компьютере или ином устройстве. Другие возможные ресурсы, которые может использовать алгоритм, — это сетевые соединения, дисковые операции и пр.

В этой книге, как и в большинстве других, посвященных алгоритмам работ, мы сосредоточиваемся только на одном ресурсе, а именно — на времени работы алгоритма. Как же судить о времени работы алгоритма? В отличие от корректности алгоритма, которая не зависит от конкретного компьютера, выполняющего алгоритм, фактическое время работы алгоритма зависит от нескольких факторов, внешних по отношению к самому алгоритму:

от скорости работы компьютера, языка программирования, на котором реализован алгоритм, компилятора или интерпретатора, который переводит программу в выполнимый код, опыта разрабатывающего программу программиста да и просто от того, чем именно параллельно выполнению вашей программы занят компьютер. Кроме того, обычно предполагается, что алгоритм выполняется на одном компьютере и все необходимые данные при этом находятся в оперативной памяти.

Если бы мы оценивали скорость работы алгоритма путем его реализации на конкретном языке программирования и измерения времени его работы на конкретном компьютере с конкретными входными данными, то это ничего не говорило бы нам о том, как быстро алгоритм выполнялся бы с входными данными другого размера (или даже просто с другими входными данными того же размера). И если бы мы хотели таким образом сравнить скорости нескольких алгоритмов, нам бы пришлось реализовывать их оба и выполнять с различными входными данными разных размеров. Так как же все-таки можно оценивать скорость алгоритма?

Ответ заключается в том, чтобы делать это с помощью объединения двух идей. Во-первых, мы определяем, как зависит время работы алгоритма от размера его входных данных. В нашем набившем оскомину примере поиска маршрута имеется некоторое представление карты дорог, размер которого зависит от количества перекрестков и количества дорог, соединяющих эти перекрестки. (Физический размер дорожной сети не имеет значения, поскольку мы можем охарактеризовать все расстояния числами, а все числа в компьютере имеют один и тот же размер; так что длина дороги не имеет никакого отношения к размеру входных данных.) В более простом примере поиска заданного элемента в списке, чтобы определить, есть он там или нет, размером входных данных является количество элементов в списке.

Во-вторых, нас интересует, как быстро с ростом размера входных данных растет время работы алгоритма — *скорость роста* времени работы. В главе 2, "Описание и оценка компьютерных алгоритмов", мы познакомимся с обозначениями, применяемыми для того, чтобы охарактеризовать время работы алгоритма. Самое интересное в нашем подходе то, что нас интересует только доминирующий член функции времени работы алгоритма, и при этом мы не учитываем коэффициенты. То есть нас интересует только ***порядок роста*** времени работы. Например, предположим, что мы выяснили, что конкретная реализация конкретного алгоритма поиска в списке из n элементов выполняется за $50n+125$ тактов процессора. Член $50n$ доминирует над членом 125 при достаточно больших значениях n — начиная с $n \geq 3$, и это доминирование растет с ростом списка размеров. Таким образом, при описании времени работы данного алгоритма мы не учитываем член более низкого порядка 125. Но что может удивить вас по-настоящему, так это то, что мы отбрасываем и коэффициент 50, таким образом, характеризуя время работы алгоритма просто как линейно растущее с увеличением размера входных данных. В качестве другого примера представим, что алгоритму требуется $20n^3+100n^2+300n+200$ тактов процессора. В этом случае мы говорим, что его время работы растет с ростом размера входных данных как n^3. Члены меньшего порядка — $100n^2$, $300n$ и 200 — становятся все менее и менее значимыми с увеличением размера входных данных n.

На практике игнорируемые нами коэффициенты имеют значение. Но они так сильно зависят от внешних факторов, что вполне возможно, что при сравнении двух алгоритмов, А и Б, которые имеют один и тот же тот же порядок роста и работают с одинаковыми входными данными, алгоритм А может выполняться быстрее, чем Б, при некотором конкретном сочетании компьютера, языка программирования, компилятора и программиста, в то время как при некотором другом сочетании указанных фактором алгоритм Б работает быстрее, чем алгоритм А. Конечно, если оба алгоритма дают верные решения и алгоритм А всегда работает вдвое быстрее, чем алгоритм Б, то при прочих равных условиях мы будем предпочитать работать с алгоритмом А. Но с точки зрения абстрактного сравнения алгоритмов мы ориентируемся по порядку роста времени работы — безо всяких украшений в виде коэффициентов или членов более низкого порядка.

Последний вопрос, который мы рассмотрим в данной главе, — "Зачем нужны алгоритмы? Почему нас, вообще говоря, должна волновать эта тема?" Ответ зависит от того, кем вы, собственно, являетесь.

Компьютерные алгоритмы для людей, не связанных с компьютерами

Даже если вы не считаете, что ваша деятельность и повседневная жизнь каким-либо образом связаны с компьютерами, все равно компьютерные алгоритмы вторгаются в вашу жизнь. Вы используете GPS? Вы что-то искали в Интернете сегодня? Любая поисковая система — будь то Google, Bing или Яндекс — использует сложные алгоритмы как для поиска в Интернете, так и для решения, в каком порядке представить результаты поиска. Вы водите современный автомобиль? Его бортовые компьютеры принимают за время поездки миллионы решений — и все на основе алгоритмов. Я мог бы продолжать, но стоит ли?

Как конечный пользователь алгоритмов вы просто обязаны узнать немного о том, как мы их проектируем, характеризуем и оцениваем. Я предполагаю, что у вас есть определенный интерес к этой теме, иначе вы бы не выбрали эту книгу и не дочитали ее до этой страницы. Тем лучше! По крайней мере, вам будет о чем поговорить на вечеринке.[1]

Компьютерные алгоритмы для компьютерщиков

Если же вы — представитель славного племени компьютерщиков, то вам и флаг в руки! Алгоритмы — столь же важная начинка вашего компьютера, как и все его железки. Вы можете купить супернавороченный компьютер и попросту выбросить деньги, если на нем не будут работать хорошие алгоритмы.

Вот пример, который иллюстрирует важность хороших алгоритмов. В главе 3, "Алгоритмы сортировки и поиска", мы встретимся с несколькими различными алгоритмами, которые сортируют список из *n* значений в порядке возрастания. Одни из этих алгоритмов

[1] *Вы можете возразить, что алгоритмы — тема для разговоров только в Кремниевой Долине, но мы, профессора информатики, не против получить приглашения на вечеринку, где не будет недостатка в таких разговорах.*

имеют время работы, растущее как n^2, у других время работы составляет только $n \lg n$. Что такое $\lg n$? Это логарифм числа n по основанию 2, или $\log_2 n$. Ученые в области информатики используют логарифмы по основанию 2 столь же часто, как математики — натуральные логарифмы $\log_e n$, для которых они даже придумали специальное обозначение $\ln n$. Далее, поскольку функция $\lg n$ является обратной к экспоненциальной функции, она растет с ростом n очень медленно. Если $n = 2^x$, то $x = \lg n$. Например, $2^{10} = 1024$, и поэтому $\lg 1024$ равен всего лишь 10. Аналогично $2^{20} = 1\,048\,576$, так что $\lg\ 1\,048\,576$ равен всего лишь 20, а то, что $2^{30} = 1\,073\,741\,824$, означает, что $\lg 1\,073\,741\,824$ равен 30.

Чтобы пример был более конкретным, рассмотрим два компьютера — А и Б. Компьютер А более быстрый, и на нем работает алгоритм сортировки, время работы которого с n значениями растет с ростом n как n^2, а более медленный компьютер Б использует алгоритм сортировки, время работы которого с n значениями растет с ростом n как $n \lg n$. Оба компьютера должны выполнить сортировку множества, состоящего из 10 миллионов чисел. (Хотя десять миллионов чисел и могут показаться огромным количеством, но если эти числа представляют собой восьмибайтовые целые числа, то входные данные занимают около 80 Мбайт памяти, что весьма немного даже для старых недорогих лэптопов.) Предположим, что компьютер А выполняет 10 миллиардов команд в секунду (что быстрее любого одного последовательного компьютера на момент написания книги), а компьютер Б — только 10 миллионов команд в секунду, так что компьютер А в 1000 раз быстрее компьютера Б. Чтобы различие стало еще бóльшим, предположим, что код для компьютера А написан самым лучшим в мире программистом на машинном языке, так что для сортировки n чисел надо выполнить $2n^2$ команд. Сортировка же на компьютере Б реализована программистом-середнячком с помощью языка высокого уровня с неэффективным компилятором, так что в результате получился код, требующий выполнения $50n \lg n$ команд. Для сортировки десяти миллионов чисел компьютеру А понадобится

$$\frac{2 \cdot \left(10^7\right)^2 \text{ команд}}{10^{10} \text{ команд/с}} = 20\,000 \text{ с},$$

т.е. более 5.5 часов, в то время как компьютеру Б потребуется

$$\frac{50 \cdot 10^7 \lg 10^7 \text{ команд}}{10^7 \text{ команд/с}} = 1163 \text{с},$$

т.е. менее 20 минут. Как видите, использование кода, время работы которого возрастает медленнее, даже при плохом компиляторе на более медленном компьютере требует более чем в 17 раз меньше процессорного времени! Если же нужно выполнить сортировку 100 миллионов чисел, то преимущество n^2-алгоритма перед $n \lg n$-алгоритмом становится еще более разительным: там, где для первого алгоритма потребуется более 23 дней, второй алгоритм справится за четыре часа. Общее правило таково: чем больше размер задачи, тем заметнее преимущество алгоритма со временем работы $n \lg n$.

Несмотря на впечатляющие достижения компьютерной техники мы видим, что общая производительность системы зависит от выбора эффективных алгоритмов ничуть не меньше, чем от выбора быстрого оборудования или эффективной операционной системы.

Такие же быстрые успехи, которые в настоящее время наблюдаются в других компьютерных технологиях, осуществляются и в области алгоритмов.

Дальнейшее чтение

По моему очень предвзятому мнению, наиболее ясным и полезным источником информации о компьютерных алгоритмах является книга *Алгоритмы: построение и анализ* [4]. Я взял из нее большую часть материала для данной книги. CLRS является гораздо более полным учебником, чем эта книга, но он предполагает, что вы хотя бы немного разбираетесь в программировании и математике. Если вы обнаружите, что математика в этой книге не вызывает у вас затруднений и вы готовы углубленно изучить ту или иную тему, то вы не сможете сделать ничего лучшего, чем обратиться к CLRS. (Конечно, это мое личное скромное мнение.)

Книга Джона Мак-Кормика (John MacCormick) *Nine Algorithms That Changed the Future* (Девять алгоритмов, изменивших будущее) [14] описывает несколько алгоритмов и связанных с ними вычислительных аспектов, которые влияют на нашу повседневную жизнь. Книга Мак-Кормика менее технична, чем данная. Если вы обнаружите, что мой подход в этой книге слишком математичен для вас, я рекомендую вам попробовать почитать книгу Мак-Кормика. Вы должны легко понимать изложенный в ней материал, даже если у вас гуманитарное образование.

В том маловероятном случае, если CLRS покажется вам слишком разбавленной, вы можете обратиться к многотомнику Дональда Кнута (Donald Knuth) *Искусство программирования* [10–13]. Хотя название может навести на мысль, что многотомник посвящен деталям написания кода, на самом деле в нем содержится блестящий, углубленный анализ алгоритмов. Но будьте осторожны: материал в *Искусстве программирования* очень сложен и переполнен серьезнейшей математикой. Кстати, если вам интересно, откуда взялось слово “алгоритм”, то Кнут утверждает, что оно является производным от имени “аль Ховаризми”, персидского математика, жившего в IX веке.

Помимо CLRS, имеется масса других отличных книг, посвященных компьютерным алгоритмам. В примечаниях к главе 1 CLRS приведен целый их список — к которому, чтобы сэкономить место здесь, я вас и отсылаю.

2... Описание и оценка компьютерных алгоритмов

В предыдущей главе вы познакомились со временем работы алгоритмов, в первую очередь — с зависимостью времени работы алгоритма как функцией от размера входных данных, и с тем, как время работы алгоритма оценивается как порядок роста этой функции. В этой главе мы немного вернемся к описанию компьютерных алгоритмов. Затем вы узнаете обозначения, которые используются при описании времени работы алгоритмов. Завершится эта глава описанием некоторых методов, которые используются для проектирования и понимания алгоритмов.

Описание компьютерных алгоритмов

У нас всегда есть возможность описания компьютерного алгоритма в виде выполнимой программы на одном из распространенных языков программирования, таком как Java, C, C++, Python или Fortran. И действительно, некоторые учебники по алгоритмам поступают именно так. Проблема при использовании реальных языков программирования для описания алгоритмов заключается в том, что при этом можно увязнуть в языковых деталях, за этими деревьями не увидев леса самого алгоритма. Другой подход, который мы приняли в CLRS, использует так называемый "псевдокод", который выглядит мешаниной из разных машинных языков программирования с естественным человеческим языком. Если вы когда-либо пользовались реальным языком программирования, вы легко разберетесь в псевдокоде. Но если вам ни разу не доводилось программировать, то псевдокод с самого начала может показаться какой-то тайнописью.

Подход, использованный мною в этой книге, заключается в том, что я пытаюсь описывать алгоритмы не для программного или аппаратного обеспечения, а для "нейронного": серого вещества, располагающегося между ушами. Я также вынужден предположить, что вы никогда не опускались до написания реальных компьютерных программ, а потому я не буду даже пытаться описывать алгоритмы с применением какого-либо реального языка программирования или даже псевдокода. Вместо этого я описываю их обычным человеческим языком, используя по возможности аналогии из реального мира. Если вдруг вы захотите реализовать описанный мною алгоритм на реальном языке программирования, я безоговорочно доверяю вам перевод моего описания в выполнимый код.

Хотя я обещал и буду по возможности стараться избегать при описании технических деталей, все же эта книга об алгоритмах *для компьютеров*, так что я вынужден использовать компьютерную терминологию. Например, компьютерные программы содержат ***процедуры*** (известные также как функции или методы в реальных языках программирования), которые указывают, как сделать то или это. Для того чтобы процедура сделала то, для чего она предназначена, ее нужно ***вызвать***. Когда мы вызываем процедуру, мы снабжаем ее входными данными (как правило, по крайней мере одним, но некоторые процедуры

обходятся и без них). Входные данные процедур мы определяем как ***параметры*** в круглых скобках после имени процедуры. Например, чтобы вычислить квадратный корень из числа, мы могли бы определить процедуру SQUARE-ROOT(x), на вход которой передается параметр x. Вызов процедуры может сгенерировать некоторые выходные данные (а может обойтись и без этого), в зависимости от того, как мы определили процедуру. Если процедура генерирует выходные данные, мы обычно рассматриваем их как нечто, передаваемое назад вызывающему процедуру коду. В компьютерных терминах это описывается как ***возврат*** значения процедурой.

Многие программы и алгоритмы работают с массивами данных. ***Массив*** группирует данные одного и того же типа в единую сущность. Вы можете думать о массиве как о таблице, которая при наличии ***индекса*** некоторой ***записи*** позволяет получить ***элемент*** массива с указанным индексом. Например, вот таблица первых пяти президентов США.

Индекс	Президент
1	Джордж Вашингтон
2	Джон Адамс
3	Томас Джефферсон
4	Джеймс Мэдисон
5	Джеймс Монро

Например, в этой таблице элемент с индексом 4 — Джеймс Мэдисон. Мы рассматриваем эту таблицу не как пять отдельных объектов, а как единую таблицу с пятью записями. Массив выглядит аналогично. Индексы в массиве представляют собой последовательные числа, которые могут начинаться с любого числа, но обычно индексы начинаются с 1[1]. Имея имя массива и индекс, мы можем объединить их с помощью квадратных скобок для указания конкретного элемента массива. Например, i-й элемент массива A обозначается как $A[i]$.

Массивы в компьютере обладают одним важным свойством: обращение к любому элементу массива выполняется за одно и то же время. Когда вы передаете компьютеру индекс i в массиве, он обращается к i-му элементу так же быстро, как и к первому, независимо от значения i.

Давайте рассмотрим наш первый алгоритм: поиск определенного значения в массиве. То есть мы имеем массив и хотим узнать, какая запись в массиве (если таковая имеется) имеет определенное значение. Чтобы понять, как можно организовать поиск в массиве, давайте представим массив как длинную полку, полную книг, и предположим, что нам надо узнать, где на полке находится книга Шолохова. Книги на полке могут быть каким-то образом упорядочены, например в алфавитном порядке по фамилии автора или в алфавитном порядке по названию, или по некоторому внутреннему библиотечному номеру. А может быть, они стоят так, как на моей полке у меня дома, где в силу моей неорганизованности книги стоят как попало.

---

[1] *Если вы программируете на Java, C или C++, то ваши массивы начинаются с нулевого элемента. Такое начало индексов с нуля отлично подходит компьютерам, но люди обычно предпочитают считать что-то, начиная с единицы.*

Если вы не можете заранее быть уверены, что книги стоят на полке в некотором порядке, как бы вы искали книгу Шолохова? Вот алгоритм, которому следовал бы я. Я бы начал с левого конца полки и посмотрел на первую книгу. Если это Шолохов, я обнаружил искомую книгу. В противном случае я бы посмотрел на следующую книгу справа, и если эта книга Шолохова, то я обнаружил искомую книгу. Если же нет, то я бы продолжил идти вправо, просматривая книгу за книгой, пока не нашел бы книгу Шолохова или пока не натолкнулся бы на правую стенку полки (и сделал бы вывод, что на этой полке книги Шолохова нет). (В главе 3, "Алгоритмы сортировки и поиска", мы узн*а*ем, как искать книги, если они стоят на полке в определенном порядке.)

Вот как мы можем описать задачу поиска в вычислительной терминологии. Будем рассматривать полку как массив книг. Крайняя слева книга находится в позиции 1, следующая книга справа он нее находится в позиции 2, и т.д. Если на полке у нас есть n книг, то крайняя справа книга находится в позиции n. Мы хотим найти номер позиции любой книги Шолохова на полке.

В качестве обобщенной вычислительной задачи мы получаем массив A (вся книжная полка, на которой мы ищем интересующую нас книгу) с n элементами (отдельными книгами), и при этом надо выяснить, присутствует ли в массиве A значение x (книга Шолохова) Если да, то мы хотим знать индекс i, такой, что $A[i] = x$ (т.е. в i-й позиции на полке стоит книга Шолохова). Мы также должны иметь возможность каким-то образом сообщить, что массив A не содержит элемент x (на полке нет книг Шолохова). Мы не ограничиваем себя предположением, что x содержится в массиве не более одного раза (возможно, на полке несколько книг Шолохова), так что если элемент x присутствует в массиве A, он может встретиться там несколько раз. Все, что мы хотим от алгоритма поиска, — *произвольный* индекс, по которому мы найдем элемент x в массиве A. Мы предполагаем, что индексы массива начинаются с 1, так что его элементами являются элементы с $A[1]$ по $A[n]$.

Если мы ищем книгу Шолохова, начиная с левого конца полки и проверяя поочередно все книги слева направо, такой метод называется ***линейным поиском***. В терминах массива в памяти компьютера мы начинаем с начала массива (первого его элемента), поочередно проверяя все его элементы ($A[1]$, затем $A[2]$, затем $A[3]$ и так далее до $A[n]$) и записывая место, где мы находим x (если мы вообще находим его).

Приведенная далее процедура LINEAR-SEARCH получает три параметра, которые в спецификации разделены запятыми.

Процедура LINEAR-SEARCH(A,n,x)

Вход:

- A: массив.
- n: количество элементов массива A, среди которых выполняется поиск.
- x: искомое значение.

Выход: либо индекс i, для которого $A[i] = x$, либо специальное значение NOT-FOUND, которое может представлять собой любой некорректный индекс массива, такой как 0 или произвольное отрицательное значение.

1. Установить значение *answer* равным NOT-FOUND.
2. Для каждого индекса *i*, пробегающего поочередно значения от 1 до *n*:
   A. Если $A[i] = x$, установить значение *answer* равным *i*.
3. В качестве выходного вернуть значение *answer*.

В дополнение к параметрам *A*, *n* и *x* процедура LINEAR-SEARCH использует ***переменную*** с именем *answer*. Процедура ***присваивает*** начальное значение NOT-FOUND переменной *answer* на шаге 1. Шаг 2 проверяет каждую запись массива от $A[1]$ до $A[n]$, не содержит ли она значение *x*. Всякий раз, когда запись $A[i]$ равна *x*, шаг 2A присваивает переменной *answer* текущее значение индекса *i*. Если *x* присутствует в массиве, то выходное значение, возвращаемое на шаге 3, представляет собой последний индекс, где встретился элемент *x*. Если же элемент *x* в массиве отсутствует, то проверка равенства на шаге 2A никогда не будет истинной, и процедура вернет значение NOT-FOUND, присвоенное переменной *answer* на шаге 1.

Прежде чем мы продолжим обсуждение линейного поиска, скажем несколько слов о том, как описать неоднократные действия, такие как на шаге 2. Такие повторяющиеся действия довольно часто встречаются в алгоритмах, например, для выполнения некоторых действий для переменной, принимающей значения из определенного диапазона. Когда мы выполняем такие неоднократные действия, это называется ***циклом***, а каждое однократное выполнение действий — ***итерацией*** цикла. Описывая цикл на шаге 2, я написал "для каждого индекса *i*, пробегающего поочередно значения от 1 до *n*". Далее вместо такого словесного описания я буду использовать запись "Для $i = 1$ до *n*", которая короче, но передает ту же структуру. Обратите внимание, что, когда я записываю цикл таким образом, ***переменная цикла*** (в данном случае — *i*) получает начальное значение (в данном случае — 1), и в каждой итерации цикла значение переменной цикла сравнивается с пределом (в данном случае — *n*). Если текущее значение переменной цикла не превышает предел, мы делаем все, что указано в ***теле*** цикла (в данном случае это строка 2A). После выполнения итерации тела цикла выполняется ***инкремент*** переменной цикла — т.е. прибавление к ней 1, — после чего выполняется возврат к заголовку цикла, где новое значение переменной цикла сравнивается с пределом. Проверка значения переменной цикла, выполнение его тела и увеличение значения переменной выполняется многократно, до тех пор, пока значение переменной цикла не превысит предел. Затем выполнение продолжается с шага, следующего после тела цикла (в данном случае с шага 3). Цикл вида "для $i = 1$ до *n*" выполняет *n* итераций и $n + 1$ проверку на превышение значения предела (поскольку значение переменной цикла превысит предел при $(n+1)$-й проверке).

Я надеюсь, для вас очевидно, что процедура LINEAR-SEARCH всегда возвращает правильный ответ. Но вы могли заметить, что эта процедура неэффективна: она продолжает поиск в массиве даже после того, как индекс *i*, для которого $A[i] = x$, уже найден. Вряд ли вы продолжаете поиск книги на полке после того, как уже нашли ее. Так что мы можем спроектировать нашу процедуру линейного поиска так, чтобы поиск прекращался, как

только он находит в массиве значение x. Далее мы считаем, что команда возврата значения немедленно прерывает выполнение процедуры и возвращает указанное значение вызывающему коду.

Процедура BETTER-LINEAR-SEARCH(A,n,x)

Вход и выход: те же, что и в LINEAR-SEARCH.

1. Для $i = 1$ до n:

   A. Если $A[i] = x$, вернуть значение i в качестве выхода процедуры.

2. Вернуть в качестве выходного значение NOT-FOUND.

Поверите ли вы в это или нет, но линейный поиск можно сделать еще более эффективным. Заметим, что при каждом выполнении итерации цикла из шага 1 процедура BETTER-LINEAR-SEARCH выполняет две проверки: проверка в первой строке для выяснения, не вышло ли значение переменной цикла за допустимые рамки (т.е. выполняется ли неравенство $i \le n$), и если не вышло, то выполняются очередная итерация цикла и проверка равенства во второй строке. В терминах поиска на книжной полке эти проверки выглядят так, как будто бы для каждой книги вы проверяли две вещи — добрались ли вы до конца полки и не является ли автором следующей книги Шолохов? В случае с полкой особых неприятностей от того, что вы пройдете немного дальше полки, не будет (конечно, если только вы не ударитесь о стену), но в случае компьютерной программы обращение к элементам массива за его концом обычно заканчивается очень плохо. Например, это может привести к повреждению данных или аварийному завершению программы.

Можно исправить алгоритм так, что вам придется выполнять для каждой книги только одну проверку. Что если бы вы точно знали, что на полке есть книга Шолохова? Тогда вы были бы твердо уверены, что найдете ее, так что до конца полки вы никогда не доберетесь. Достаточно просто проверять все книги по очереди, пока не встретитесь с книгой Шолохова.

Но представим, что вы ищете книгу на чужой полке или на своей, но не помните, на какую именно полку засунули книги Шолохова — словом, вы не уверены, что на полке есть требуемая книга. Вот что вы можете сделать в этом случае. Возьмите пустую коробку размером с книгу и на узкой ее стороне (имитирующей корешок книги) напишите "*Шолохов. Поднятая целина*". Замените крайнюю справа книгу этой коробкой, после чего при поиске вдоль полки слева направо вам нужно будет проверять только автора книги, но не то, последняя ли эта книга на полке, потому что теперь вы точно знаете, что найдете что-то, на чем написано "Шолохов". Единственный вопрос заключается в том, нашли ли вы настоящую книгу или ее заменитель. Если это заменитель, значит, книги Шолохова на полке нет. Но такую достаточно простую проверку надо выполнить только один раз, в конце поиска, а не для каждой книги на полке.

Имеется еще одна деталь, о которой надо упомянуть: что если единственная книга Шолохова на вашей полке была крайней справа? Если вы заменили ее пустой коробкой,

поиск завершится, когда вы ее найдете, и вы можете ошибочно заключить, что искомой книги на полке нет. Поэтому следует выполнить еще одну проверку — для выявления такой возможности. Но это тоже только одна проверка, не выполняемая для каждой книги на полке.

В терминах компьютерного алгоритма мы помещаем искомое значение x в последнюю позицию $A[n]$ после сохранения содержимого $A[n]$ в другой переменной. Когда мы находим x, мы выполняем проверку, *действительно* ли мы его нашли. Значение, которое мы помещаем в массив, называется ***ограничителем***, но его можно рассматривать как пустой ящик.

Процедура SENTINEL-LINEAR-SEARCH(A,n,x)

Вход и выход: те же, что и в LINEAR-SEARCH.

1. Сохранить $A[n]$ в переменной *last* и поместить x в $A[n]$.
2. Установить i равным 1.
3. Пока $A[i] \neq x$, выполнять следующие действия:
   А. Увеличить i на 1.
4. Восстановить $A[n]$ из переменной *last*.
5. Если $i < n$ или $A[n] = x$, вернуть значение i в качестве выхода процедуры.
6. В противном случае вернуть в качестве возвращаемого значения NOT-FOUND.

Шаг 3 представляет собой цикл, но цикл, в котором не происходит увеличения переменной-счетчика. Вместо этого итерации цикла выполняются до тех пор, пока выполняется условие цикла; в данном случае это условие $A[i] \neq x$. Цикл работает следующим образом. Сначала выполняется проверка условия цикла (в данном случае — $A[i] \neq x$), и если оно истинно, то выполняется тело цикла (в данном случае — строка 3А, которая увеличивает переменную i на 1). Затем выполняется возврат в начало цикла, и, если условие цикла истинно, снова выполняется тело цикла. И так проверка условия с последующим выполнением тела цикла продолжается до тех пор, пока при очередном выполнении условие не окажется ложным. Тогда алгоритм продолжается с шага, следующего за телом цикла (в данном случае — со строки 4).

Процедура SENTINEL-LINEAR-SEARCH оказывается немного более сложной по сравнению с первыми двумя процедурами линейного поиска. Поскольку она помещает x в $A[n]$ на шаге 1, мы гарантируем, что $A[i]$ будет равно x при некоторой проверке на шаге 3. Как только это произойдет, мы прекращаем выполнение цикла на шаге 3, и индекс i больше изменяться не будет. Прежде чем мы сделаем что-либо еще, на шаге 4 мы восстанавливаем исходное значение $A[n]$. (Моя мать всегда учила меня, попользовавшись чем-то, положить все назад, откуда взял.) Затем мы должны определить, действительно ли мы нашли в массиве x. Поскольку мы поместили x в последний элемент, $A[n]$, мы знаем, что если мы

нашли x в $A[i]$, где $i < n$, то мы действительно нашли x и можно смело возвращать индекс, хранящийся в переменной i. Но что делать, если мы нашли x в $A[n]$? Это означает, что мы не нашли x до $A[n]$, так что нам надо проверить, действительно ли $A[n]$ равен x. Если это так, то мы возвращаем индекс n, который в этом случае равен i, но если $A[n] \neq x$, то следует возвратить значение NOT-FOUND. Эти проверки выполняются на шаге 5, и, если x изначально находился в массиве, на этом же шаге возвращается корректное значение индекса. Если же x найден только потому, что это значение было помещено в массив на шаге 1, то шаг 6 вернет значение NOT-FOUND. Хотя процедура SENTINEL-LINEAR-SEARCH и выполняет две проверки по окончании цикла, на каждой итерации выполняется только одна проверка, так что эта процедура оказывается более эффективной, чем процедуры LINEAR-SEARCH и BETTER-LINEAR-SEARCH.

Описание времени работы алгоритма

Вернемся к процедуре LINEAR-SEARCH на с. 25 и рассмотрим время ее работы. Вспомним, что мы хотим выразить время работы алгоритма как функцию от размера его входных данных. В этой процедуре входными данными являются массив A из n элементов, а также значение n и искомое значение x. Значения n и x не играют никакой роли при достаточно большом размере массива (в конце концов, n представляет собой всего лишь одно целое число, а x имеет размер одного элемента массива), так что можно считать, что размер входных данных равен n, числу элементов в массиве A.

Мы должны сделать несколько простых предположений о том, сколько времени занимают простые операции. Мы будем предполагать, что каждая отдельная операция — арифметическая (например, сложения, вычитания, умножения или деления), сравнение, присваивание значения переменной, индексация элемента массива, вызов процедуры или возврат из нее — выполняется за некоторое фиксированное время, не зависящее от размера входных данных[2]. Время выполнения может варьироваться от операции к операции, так что деление может занять больше времени, чем сложение, но когда шаг алгоритма состоит только из простых операций, каждое отдельное выполнение этого шага занимает некоторое постоянное количество времени. Поскольку выполняемые операции выполняются от шага к шагу, а также по внешним причинам, перечисленным в подразделе "Использование ресурсов" на с. 17, время выполнения также может отличаться от шага к шагу. Будем считать, что каждое выполнение шага i требует времени t_i, где t_i — некоторые константы, не зависящие от n.

Конечно, необходимо учитывать, что некоторые шаги выполняются несколько раз. Например, шаги 1 и 3 выполняются только один раз, но что можно сказать о шаге 2?

[2] *Если вы немного знакомы с архитектурой реальных компьютеров, то можете знать, что время обращения к конкретной переменной или элементу массива не обязательно фиксированное, поскольку может зависеть от того, где находится переменная или элемент массива — в кеш-памяти, основной оперативной памяти или в виртуальной памяти на диске. Некоторые сложные модели компьютеров учитывают эти вопросы, но зачастую вполне достаточно считать, что все переменные и элементы массивов находятся в основной памяти компьютера и что время обращения к любому из них одинаково.*

Мы должны проверять i на равенство n в общей сложности $n+1$ раз: n раз выполняется условие $i \le n$ и один раз — $i = n+1$, и мы выходим из цикла. Шаг 2 выполняется ровно n раз, по одному разу для каждого из значений i от 1 до n. Мы не знаем заранее, сколько раз будем присваивать переменной *answer* значение i; это количество может принимать любые значения от 0 (если x нет в массиве) до n раз (если каждый элемент массива равен x). Если мы собираемся быть предельно точны в нашем подсчете — а обычно мы не будем столь придирчивы, — то должны считать, что на шаге 2 выполняются два разных действия, которые выполняются различное количество раз: сравнение i с n выполняется $n+1$ раз, но увеличение i выполняется только n раз. Разделим время выполнения строки 2 на t_2' для выполнения проверки и t_2'' для увеличения значения i. Аналогично разделим время выполнения строки 2A на t_{2A}', требующееся для проверки $A[i] = x$, и t_{2A}'', необходимое для присваивания переменной *answer* значения i. Тогда время выполнения процедуры LINEAR-SEARCH находится где-то между

$$t_1 + t_2' \cdot (n+1) + t_2'' \cdot n + t_{2A}' \cdot n + t_{2A}'' \cdot 0 + t_3$$

и

$$t_1 + t_2' \cdot (n+1) + t_2'' \cdot n + t_{2A}' \cdot n + t_{2A}'' \cdot n + t_3.$$

Перепишем теперь эти границы иначе, собирая вместе члены с n, и отдельно — остальные члены, и при этом сразу станет очевидно, что время выполнения находится между ***нижней границей***

$$(t_2' + t_2'' + t_{2A}') \cdot n + (t_1 + t_2' + t_3)$$

и ***верхней границей***

$$(t_2' + t_2'' + t_{2A}' + t_{2A}'') \cdot n + (t_1 + t_2' + t_3).$$

Заметим, что обе границы имеют вид $c \cdot n + d$, где c и d — константы, не зависящие от n. Время работы процедуры LINEAR-SEARCH ограничено линейной функцией от n как снизу, так и сверху.

Для обозначения того факта, что время работы ограничено как сверху, так и снизу некоторыми (возможно, разными) линейными функциями от n, используются специальные обозначения. Мы записываем, что время работы алгоритма равно $\Theta(n)$. Это греческая буква тета, и мы произносим эту запись как "тета от n" или просто как "тета n". Как я и обещал в главе 1, "Что такое алгоритмы и зачем они нужны", это обозначение отбрасывает член меньшего порядка $(t_1 + t_2' + t_3)$ и коэффициенты при n ($(t_2' + t_2'' + t_{2A}')$ для нижней границы и $(t_2' + t_2'' + t_{2A}' + t_{2A}'')$ для верхней границы). Хотя при записи времени работы алгоритма как $\Theta(n)$ мы теряем точность, зато получаем преимущество в том, что у нас остается ясно выраженный порядок роста и отброшены все несущественные детали.

Θ-обозначения применимы к функциям в общем случае, а не только для описания времени работы алгоритмов, и применимы не только к линейным функциям. Идея заключается в том, что если у нас имеются две функции, $f(n)$ и $g(n)$, то мы говорим, что $f(n)$ представляет собой $\Theta(g(n))$, если $f(n)$ при достаточно больших n отличается от $g(n)$ на постоянный множитель. Так что мы можем сказать, что время работы процедуры LINEAR-SEARCH при достаточно больших n отличается от n на постоянный множительИмеется пугающее техническое определение Θ-обозначения, но, к счастью, нам редко требуется обращаться к нему, чтобы использовать Θ-обозначения. Мы просто сосредоточиваемся

на доминирующем члене, отбрасывая члены более низкого порядка и постоянные множители. Например, функция $n^2/4+100n+50$ представляет собой $\Theta(n^2)$; здесь мы опускаем члены более низкого порядка $100n$ и 50, а также постоянный множитель 1/4. Хотя при небольших значениях n члены низкого порядка будут превышать $n^2/4$, как только n превысит 400, член $n^2/4$ превысит $100n+50$. При $n=1000$ доминирующий член $n^2/4$ равен 250 000, в то время как члены более низкого порядка $100n+50$ равны только 100 050; для $n=2000$ разница становится еще большей — 1 000 000 против 200 050. В мире алгоритмов мы немного злоупотребляем обозначениями и просто пишем равенство $f(n)=\Theta(g(n))$, так что мы можем написать $n^2/4+100n+50=\Theta(n^2)$.

Давайте теперь рассмотрим время работы процедуры BETTER-LINEAR-SEARCH на с. 27. от случай немного сложнее процедуры LINEAR-SEARCH, потому что мы не знаем заранее, сколько раз будет выполняться итерация цикла. Если $A[1]=x$, то она будет выполнена только один раз. Если x в массиве отсутствует, то цикл будет выполняться все максимально возможные n раз. Каждая итерация цикла выполняется за некоторое постоянное время, так что мы можем сказать, что *в наихудшем случае* процедура BETTER-LINEAR-SEARCH требует время $\Theta(n)$ при поиске в массиве из n элементов. Почему "в наихудшем случае"? Потому что, поскольку мы хотим, чтобы алгоритмы работали как можно быстрее, наихудший случай осуществляется, когда время работы алгоритма максимально среди времен для всех возможных входных данных.

В наилучшем случае, $A[1]=x$, процедура BETTER-LINEAR-SEARCH требует только постоянного количества времени: она устанавливает значение переменной i равным 1, проверяет, что $i \le n$, убеждается, что выполняется условие $A[i]=x$, и процедура возвращает значение i, равное 1. Это количество времени не зависит от n. Мы говорим, что время работы BETTER-LINEAR-SEARCH *в наилучшем случае* представляет собой $\Theta(1)$, потому что в этом случае время работы отличается от 1 на некоторый постоянный множитель. Другими словами, время работы в наилучшем случае представляет собой константу, которая не зависит от n.

Итак, мы видим, что не можем использовать Θ-обозначения для выражения, охватывающего все варианты времени работы процедуры BETTER-LINEAR-SEARCH. Мы не можем сказать, что ее время работы всегда представляет собой $\Theta(n)$, потому что в наилучшем случае оно является $\Theta(1)$. И мы не можем сказать, что время работы всегда является $\Theta(1)$, потому что в наихудшем случае оно представляет собой $\Theta(n)$. Однако мы можем сказать, что линейная функция от n — верхняя граница во всех случаях, и у нас есть обозначение для этого факта — $O(n)$. Когда мы записываем это обозначение, мы говорим "О большое от n" или даже просто "о от n". Функция $f(n)$ является $O(g(n))$, если при достаточно больших n функция $f(n)$ ограничена сверху $g(n)$, умноженной на некоторую константу. Мы вновь немного злоупотребляем обозначениями и используем знак равенства, записывая $f(n)=O(g(n))$. Для процедуры BETTER-LINEAR-SEARCH можно записать, что ее временем работы во всех случаях является $O(n)$; хотя время ее работы может быть лучше линейной функции n, хуже нее оно никогда не будет.

Мы используем O-обозначения для указания того факта, что время работы никогда не будет хуже, чем константа, умноженная на некоторую функцию от n. Но как указать, что

время работы никогда *не лучше*, чем константа, умноженная на некоторую функцию от n? Это нижняя граница, и мы используем Ω-обозначения, которые зеркальны O-обозначениям: функция $f(n)$ представляет собой $\Omega(g(n))$, если, когда n становится достаточно большим, $f(n)$ ограничена снизу некоторой константой, умноженной на $g(n)$. Мы говорим, что $f(n)$ представляет собой "омега большое от $g(n)$" или даже просто "омега от $g(n)$", и записываем это как $f(n) = \Omega(g(n))$. Поскольку O-обозначения дают верхнюю границу, Ω-обозначение дает нижнюю границу, а Θ -обозначение дает и верхнюю, и нижнюю границы, можно заключить, что функция $f(n)$ является $\Theta(g(n))$ тогда и только тогда, когда $f(n)$ одновременно является и $O(g(n))$, и $\Omega(g(n))$.

Мы можем высказать общее утверждение о нижней границе времени работы процедуры Better-Linear-Search: во всех случаях это $\Omega(1)$. Конечно, это трогательно слабое утверждение, поскольку любой алгоритм при любых входных данных работает по крайней мере какое-то постоянное время. Мы небудем часто обращаться к Ω-обозначениям, но иногда они могут и пригодиться.

Общим названием для Θ-, O- и Ω-обозначений является ***асимптотические обозначения***. Дело в том, что эти обозначения описывают рост функции при асимптотическом приближении ее аргумента к бесконечности. Все эти асимптотические обозначения дают возможность отбросить члены более низкого порядка и константные множители, так что мы можем игнорировать утомительные детали и сосредоточиться на действительно важном вопросе — как функция растет с ростом n.

Теперь вернемся к процедуре Sentinel-Linear-Search на с. 28. Так же, как и в случае процедуры Better-Linear-Search, каждая итерация цикла выполняется за некоторое постоянное время, и при этом может выполняться от 1 до n итераций. Ключевое различие между процедурами Sentinel-Linear-Search и Better-Linear-Search заключается в том, что время выполнения одной итерации в Sentinel-Linear-Search меньше, чем время выполнения одной итерации в Better-Linear-Search. Обе процедуры требуют линейного времени в наихудшем случае, но постоянный множитель в Sentinel-Linear-Search оказывается лучшим. Хотя мы и ожидаем, что на практике поиск с помощью процедуры Sentinel-Linear-Search будет быстрее поиска с помощью Better-Linear-Search, он будет быстрее всего лишь на постоянный множитель. Если выразить время работы этих процедур с помощью асимптотических обозначений, мы получим один и тот же результат: $\Theta(n)$ в наихудшем случае, $\Theta(1)$ в наилучшем случае и $O(n)$ во всех случаях.

Инварианты циклов

Для всех трех вариантов линейного поиска легко увидеть, что каждый из них дает правильный ответ. Зачастую увидеть это немного сложнее. Для этого имеется гораздо больше методов, чем я могу рассмотреть в этой книге.

Один из распространенных методов показа правильности алгоритма использует ***инвариант цикла***: утверждение, для которого мы демонстрируем истинность в начале каждой итерации цикла. Чтобы инвариант цикла мог помочь доказать корректность алгоритма, мы должны показать выполнение трех его свойств.

Инициализация. Инвариант цикла истинен перед первой итерацией цикла.

Сохранение. Если инвариант цикла истинен перед итерацией цикла, он остается истинным и после нее.

Завершение. Цикл завершается, а после его завершения инвариант цикла вместе с причиной завершения цикла дают нам искомую цель работы алгоритма.

В качестве примера приведем инвариант цикла для процедуры BETTER-LINEAR-SEARCH.

В начале каждой итерации на шаге 1, если x имеется в массиве A, то он находится в ***подмассиве*** (непрерывной части массива) с элементами от $A[i]$ до $A[n]$.

Чтобы показать, что если процедура возвращает индекс, отличный от NOT-FOUND, то этот индекс корректен, нам даже не нужен инвариант цикла: единственная причина возврата индекса i на шаге 1A — выполнение равенства $A[i] = x$. Вместо этого давайте воспользуемся инвариантом цикла, чтобы показать, что если процедура возвращает на шаге 2 значение NOT-FOUND, то x в массиве отсутствует.

Инициализация. Изначально $i = 1$, так что подмассив в инварианте цикла представляет собой все элементы цикла от $A[1]$ до $A[n]$, т.е. весь массив полностью.

Сохранение. Предположим, что в начале итерации для некоторого значения i, если x имеется в массиве A, то он присутствует в подмассиве от элемента $A[i]$ до элемента $A[n]$. Если эта итерация выполняется без возврата значения, то мы знаем, что $A[i] \neq x$, и, следовательно, можем утверждать, что если x присутствует в массиве A, то он находится в подмассиве от элемента $A[i+1]$ до элемента $A[n]$. Поскольку i перед следующей итерацией увеличивается на единицу, инвариант цикла перед очередной итерацией будет выполняться.

Завершение. Цикл должен завершаться либо потому, что процедура вернет значение на шаге 1A, либо потому, что выполнится условие $i > n$. Мы уже рассмотрели случай, когда цикл завершается из-за возврата значения на шаге 1A.

Чтобы разобраться со случаем завершения цикла по условию $i > n$, рассмотрим обратную форму утверждения (его контрапозицию). ***Контрапозицией*** утверждения "если А, то Б" является утверждение "если не Б, то не А". Контрапозиция истинна тогда и только тогда, когда истинно исходное утверждение. Контрапозицией инварианта цикла является утверждение "если x отсутствует в подмассиве от $A[i]$ до $A[n]$, то его нет и в массиве A".

Теперь, когда $i > n$, подмассив от $A[i]$ до $A[n]$ пуст, в нем нет ни одного элемента, так что он никак не может содержать x. В силу контрапозиции инварианта цикла x отсутствует в массиве A, так что возврат значения NOT-FOUND на шаге 2 является корректным.

Однако как много рассуждений требуется провести для такого в действительности простого цикла! Неужели мы каждый раз при рассмотрении нового алгоритма должны

проходить через все эти мучения? Я считаю, что нет, но есть ряд ученых в области информатики, которые настаивают на таких строгих рассуждениях для каждого отдельного цикла. Когда я пишу реальную программу, я замечаю, что бо́льшую часть времени написания цикла его инвариант прячется где-то на задворках моей памяти. Он может прятаться в голове так далеко, что я даже не уверен, есть ли он у меня, но если бы это потребовалось, я бы тут же смог его сформулировать. Хотя большинство из нас согласится, что инвариант цикла является излишеством для понимания работы простого цикла в процедуре Better-Linear-Search, в действительности инварианты цикла могут очень помочь, когда надо понять, как более сложные циклы делают правильные вещи.

Рекурсия

С помощью ***рекурсии*** мы решаем задачу путем решения меньших экземпляров этой же задачи. Вот мой любимый канонический пример рекурсии: вычисление $n!$ ("n факториал"), который определен для неотрицательных значений n как $n! = 1$ при $n = 0$, и

$$n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdots 3 \cdot 2 \cdot 1$$

при $n \geq 1$. Например, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$. Заметим, что

$$(n-1)! = (n-1) \cdot (n-2) \cdot (n-3) \cdots 3 \cdot 2 \cdot 1,$$

так что

$$n! = n \cdot (n-1)!$$

при $n \geq 1$. Мы определяем $n!$ через "меньшую" задачу, а именно $(n-1)!$. Таким образом, мы можем записать рекурсивную процедуру для вычисления $n!$ следующим образом.

Процедура Factorial(n)

Вход: целое число $n \geq 0$.

Выход: значение $n!$.

1. Если $n = 0$, вернуть 1 в качестве возвращаемого значения.
2. В противном случае вернуть n, умноженное на значение, которое возвращает рекурсивно вызванная процедура Factorial($n-1$).

Я записал шаг 2 довольно громоздким способом. Вместо этого можно было бы просто написать "в противном случае вернуть $n \cdot$Factorial($n-1$)", использовав возвращаемое значение рекурсивного вызова в большем арифметическом выражении.

Чтобы рекурсия работала, должны выполняться два свойства. Во-первых, должен существовать один или несколько ***базовых случаев***, когда вычисления проводятся непосредственно, без рекурсии. Во-вторых, каждый рекурсивный вызов процедуры должен быть ***меньшим экземпляром той же самой задачи***, так что в конечном итоге будет достигнут один из базовых случаев. В случае процедуры Factorial базовый случай осуществляется при $n = 0$, и каждый рекурсивный вызов выполняется для значения n, уменьшенного на 1. В случае, когда исходное значение n больше нуля, рекурсивные вызовы в конечном счете придут к базовому случаю.

Доказательство того, что рекурсивный алгоритм корректно работает, на первый взгляд, может показаться очень простой задачей. Ключевой идеей является то, что мы принимаем в качестве гипотезы утверждение о том, что каждый рекурсивный вызов дает правильный результат. Если же рекурсивные вызовы выполняют вычисления правильно, то доказать корректность процедуры очень легко. Вот как мы могли бы доказать, что процедура FACTORIAL возвращает правильное значение факториала. Очевидно, что при $n = 0$ она возвращает 1, которая является верным значением, так как $0! = 1$. Предположим, что при $n \geq 1$ рекурсивный вызов FACTORIAL($n-1$) возвращает корректное значение, т.е. $(n-1)!$. Затем процедура умножает это значение на n, тем самым вычисляя значение $n!$, которое и возвращается процедурой.

А вот пример, в котором рекурсивный вызов выполняется не для меньшего, а для большего экземпляра задачи, несмотря на корректность математической постановки задачи. При $n \geq 0$ совершенно справедливо равенство $n! = (n+1)!/(n+1)$. Но построенная по этой формуле рекурсивная процедура не дает правильный ответ при $n \geq 1$.

Процедура BAD-FACTORIAL(n)

Вход и выход: те же, что и для FACTORIAL.

1. Если $n = 0$, вернуть 1 в качестве возвращаемого значения.
2. В противном случае вернуть BAD-FACTORIAL$(n+1)/(n+1)$.

Если бы мы вызвали BAD-FACTORIAL(1), этот вызов сгенерировал бы вызов BAD-FACTORIAL(2), тот, в свою очередь, — вызов BAD-FACTORIAL(3) и т.д. В результате процедура никогда бы не добралась до базового случая, когда $n = 0$. Если бы вы реализовали эту процедуру на реальном языке программирования и запустили ее на реальном компьютере, то быстро бы увидели что-то вроде сообщения "ошибка переполнения стека".

Зачастую можно переписать алгоритм с циклом в рекурсивном стиле. Например, вот как выглядит линейный поиск без ограничителя в рекурсивном варианте.

Процедура RECURSIVE-LINEAR-SEARCH(A,n,i,x)

Вход: тот же, что и для LINEAR-SEARCH, но с дополнительным параметром i.

Выход: индекс элемента, равного x, в подмассиве от элемента $A[i]$ до $A[n]$, или значение NOT-FOUND, если x в этом подмассиве отсутствует.

1. Если $i > n$, вернуть NOT-FOUND.
2. В противном случае ($i \leq n$), если $A[i] = x$, вернуть i.
3. В противном случае ($i \leq n$ и $A[i] \neq x$), вернуть RECURSIVE-LINEAR-SEARCH($A,n,i+1,x$).

Здесь подзадача заключается в поиске элемента, равного x, в подмассиве от элемента $A[i]$ до элемента $A[n]$. Базовый случай осуществляется на шаге 1, когда подмассив пуст, т.е. когда $i > n$. Поскольку на каждом шаге значение i увеличивается на единицу при каж-

дом рекурсивном вызове, если ни один из вызовов не вернет значение i на шаге 2, в конечном итоге i превысит n, и будет достигнут базовый случай рекурсии.

Дальнейшее чтение

Главы 2 и 3 CLRS [4] охватывают большую часть материала данной главы. Более ранний учебник по алгоритмам Ахо (Aho), Хопкрофта (Hopcroft) и Ульмана (Ullman) [1] оказал особое влияние на применение асимптотических обозначений для анализа алгоритмов. Что касается доказательств корректности работы программ, то, если вы хотите углубиться в эту область, познакомьтесь с книгами Гриза (Gries) [8] и Митчелла (Mitchell) [15].

3...Алгоритмы сортировки и поиска

В главе 2 мы видели три варианта линейного поиска в массиве. Можно ли как-то улучшить имеющиеся алгоритмы? Ответ: это зависит от обстоятельств. Если мы ничего не знаем о порядке элементов в массиве, то ничего лучшего мы не добьемся. В худшем случае мы должны просмотреть все n элементов, потому что, если мы не смогли найти значение, которое мы ищем, среди первых $n-1$ элементов, оно может оказаться в последнем, n-м элементе. Таким образом, мы не можем достичь времени работы в наихудшем случае лучшего, чем $\Theta(n)$, если ничего неизвестно о порядке элементов в массиве.

Предположим, однако, что массив отсортирован в неубывающем порядке, т.е. каждый элемент массива меньше или равен элементу, следующему в массиве за ним, согласно некоторому определению отношения "меньше, чем". В этой главе мы увидим, что если массив отсортирован, то мы можем использовать простой метод, известный как бинарный поиск, чтобы найти нужный элемент в массиве из n элементов за время всего лишь $O(\lg n)$. Как мы видели в главе 1, "Что такое алгоритмы и зачем они нужны", значение $\lg n$ растет с ростом n очень медленно, и поэтому бинарный поиск превосходит линейный в наихудшем случае.[1]

Что это означает — что один элемент меньше другого? Если элементы являются числами, это очевидно. Если элементы являются строками символов текста, можно говорить о ***лексикографическом порядке***: один элемент меньше другого, если в словаре он должен располагаться первым. Когда элементы представляют собой другие виды данных, мы должны определить, что означает понятие "меньше". При наличии некоторого точного понятия "меньше, чем" можно определить, отсортирован ли массив.

Вспомним пример книжной полки из главы 2, "Описание и оценка компьютерных алгоритмов". Мы могли бы рассортировать книги в алфавитном порядке по фамилии автора, в алфавитном порядке по названию или, если в библиотеке имеется каталог, по номерам в каталоге. В этой главе мы будем считать, что книги на полке отсортированы в алфавитном порядке по автору, слева направо. На полке может содержаться более одной книги одного автора — возможно, у вас несколько книг Уильяма Шекспира. Если требуется искать не просто книгу Шекспира, а конкретную его книгу, то будем считать, что если на полке стоят две книги одного и того же автора, то та из книг, название которой идет в алфавитном порядке первым, должна находиться левее. Может оказаться и так, что все, что нас интересует, — это имя автора, так что при поиске нас устроит любая книга Шекспира. Информация, которую мы сопоставляем с книгой при поиске, называется ***ключом***. В нашем примере с книжной полкой ключом является только имя автора, а не сочетание сначала имени автора, а затем названия книги при наличии нескольких книг одного автора.

---

[1] *Если вы не компьютерщик и пропустили подраздел "Компьютерные алгоритмы для компьютерщиков" главы 1, "Что такое алгоритмы и зачем они нужны", можете вернуться к с.19 и прочесть, что такое логарифмы.*

Каким же образом нам получить отсортированный массив? В этой главе мы познакомимся с четырьмя алгоритмами — сортировка выбором, сортировка вставкой, сортировка слиянием и быстрая сортировка, — предназначенными для сортировки массива, и применением каждого из этих алгоритмов к нашей книжной полке. Каждый алгоритм сортировки имеет свои достоинства и свои недостатки, и в конце этой главы мы сравним их. Все алгоритмы сортировки, которые приведены в данной главе, имеют время работы в худшем случае либо $\Theta(n^2)$, либо $\Theta(n \lg n)$. Таким образом, если вы собираетесь выполнить лишь несколько поисков, то лучше остановиться на линейном поиске. Но если вы собираетесь выполнять поиск много раз, то может иметь смысл сначала отсортировать массив, а затем применять бинарный поиск.

Сортировка — важная задача сама по себе, а не только как шаг предварительной обработки данных для бинарного поиска. Подумайте обо всех данных, которые должны быть отсортированы, — таких как записи в телефонной книге, чеки в ежемесячной выписке банка да и просто результаты работы поисковой веб-системы, отсортированные по уровню релевантности. Кроме того, сортировка часто является отдельным шагом в другом алгоритме. Например, в компьютерной графике объекты часто перекрывают один другой. Программа, которая отображает объекты на экране, должна отсортировать их в соответствии с отношением "находится выше", чтобы затем иметь возможность изображать их один за другим снизу вверх.

Прежде чем идти дальше, пару слов о том, что такое сортировка. В дополнение к ключу (который при сортировке мы будем называть ***ключом сортировки***) сортируемые элементы обычно содержат некоторую информацию, которую мы называем ***сопутствующими данными***. Сопутствующие данные представляют собой информацию, которая связана с ключом сортировки и перемещается при перемещении элементов вместе с ключом. В нашем примере с книжной полкой ключом сортировки является фамилия автора, а сопутствующими данными — сама книга.

Своим студентам я поясняю концепцию сопутствующих данных следующим образом, который заставляет их быстро понять эту тему. Я показываю им таблицу успеваемости, отсортированную по столбцу фамилий студентов (ключом сортировки являются фамилии, а сопутствующими данными — успеваемость). Если я теперь отсортирую таблицу так, что ключом сортировки будет успеваемость (в порядке ее убывания), а фамилии студентов — сопутствующими данными, и при этом я *не буду* перемещать фамилии, оставив их в первом столбце и сортируя только столбец с данными об успеваемости... Думаю, вы понимаете, какую реакцию это вызовет у аудитории. Обычно студенты с фамилиями в начале списка оказываются очень довольными, особенно по сравнению со студентами с фамилиями на последние буквы алфавита. Зато понимание, что такое сопутствующие данные, гарантируется!

Вот некоторые другие примеры ключей сортировки и сопутствующих данных. В телефонной книге ключом сортировки является фамилия и имя абонента, а сопутствующими данными — адрес и номер телефона. В поисковой системе ключ сортировки представляет собой меру релевантности, а сопутствующими данными является URL веб-страницы, а также любая иная информация о странице, которую хранит поисковая система.

При работе с массивами в этой главе мы действуем так, как будто каждый элемент содержит только ключ сортировки. При реализации любого из приведенных здесь алгоритмов сортировки следует убедиться, что при перемещении ключа сортировки перемещаются и сопутствующие данные, или по крайней мере указатель на них.

Чтобы аналогия с книжной полкой была применима к компьютерным алгоритмам, необходимо, чтобы полка и книги обладали двумя не слишком реалистичными свойствами. Во-первых, все книги на полке должны иметь один и тот же размер, потому что в массиве в компьютере все элементы массива имеют одинаковый размер. Во-вторых, все позиции книг на полке можно пронумеровать числами от 1 до n, и каждую такую позицию мы будем называть ***слотом***. Слот 1 — крайний слева, а слот n — крайний справа. Как вы, наверное, догадались, каждому слоту на полке соответствует запись массива.

Я также хочу разобраться со словом “сортировка”. В обыденной речи сортировка может означать нечто, существенно отличное от того, что это слово означает в вычислительной технике. Словарь на моем компьютере определяет термин “сортировка” как “систематическая организация в группах; разделение в соответствии с типом, классом и т.д.”. Так что этим термином, например, можно назвать раскладывание по полкам шкафа одежды: рубашки — в одно место, галстуки — в другое и т.д. В мире компьютерных алгоритмов сортировка означает размещение элементов в некотором строго определенном порядке, а “систематическая организация в группах” называется “группировкой”.

Бинарный поиск

Прежде чем перейти к некоторым алгоритмам сортировки, давайте рассмотрим бинарный поиск, который требует, чтобы массив, в котором он выполняется, был отсортирован. Бинарный поиск имеет то преимущество, что для поиска в массиве из n элементов требуется время $O(\lg n)$.

В нашем примере с книжной полкой мы работаем с книгами, уже отсортированными по автору слева направо. Давайте, используя в качестве ключа фамилию автора книги, поищем любую книгу Маяковского. Конечно, буква “М” находится примерно в средине алфавита, но даже если бы мы искали книгу Шолохова, это еще не гарантировало бы, что она будет близко к правому концу полки, — в конце концов, на полке может оказаться большое количество книг Шукшина или Эренбурга.

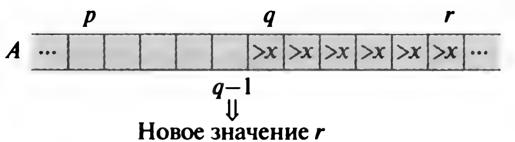
Словом, не будем полагаться на удачу и будем искать книгу Маяковского следующим образом. Перейдем к слоту, находящемуся на полке ровно посредине, возьмем находящуюся там книгу и посмотрим, кто ее автор. Предположим, что вы нашли книгу Ефремова. Это не просто не та книга, которую мы ищем; поскольку книги отсортированы по автору, мы точно знаем, что среди книг левее найденной книги Ефремова нужной нам быть не может. Просмотрев только одну книгу, мы исключили из рассмотрения половину книг на полке! Любая книга Маяковского должна быть на правой половине полки. Так что теперь мы найдем слот посредине правой половины полки и посмотрим, какая книга находится в этом месте. Предположим, что это книга Льва Толстого. Далее найдем книгу в слоте посреди третьей четверти полки, в которой только и может располагаться книга Маяковского. Если это Маяковский, мы нашли то, что искали. Если нет, мы снова можем исключить половину

оставшихся книг. В конце концов мы либо найдем книгу Маяковского, либо доберемся до такой мелкой части полки, в которой уже не будет ни одного слота, который мог бы содержать искомую книгу. В последнем случае можно заключить, что книги Маяковского на полке нет.

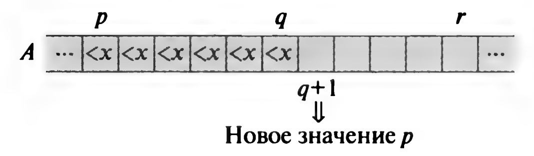
В компьютере мы выполняем бинарный поиск в массиве. В любой момент мы рассматриваем только подмассив, т.е. часть массива между двумя индексами (включительно). Назовем их p и r. Первоначально $p = 1$ и $r = n$, так что в начале работы подмассив совпадает со всем массивом. Мы многократно делим подмассив пополам, до тех пор, пока не произойдет одно из двух событий: либо мы найдем искомое значение, либо подмассив окажется пустым (т.е. p станет больше, чем r). Именно это многократное деление подмассива пополам и обеспечивает время работы алгоритма, равное $O(\lg n)$.

Рассмотрим работу бинарного поиска немного более подробно. Предположим, что мы ищем значение x в массиве A. На каждом шаге мы рассматриваем только подмассив, начинающийся с элемента $A[p]$ и заканчивающийся элементом $A[r]$. Поскольку нам придется немного поработать с подмассивами, введем для такого подмассива обозначение $A[p..r]$. На каждом шаге мы вычисляем средину q рассматриваемого подмассива, вычисляя среднее p и r и отбрасывая дробную часть, если таковая имеется: $q = \lfloor (p+r)/2 \rfloor$ (здесь мы используем функцию "пол" $\lfloor\ \rfloor$ для удаления дробной части числа. При реализации этой операции на языках программирования Java, C или C++ можно просто использовать целочисленное деление, при котором дробная часть отбрасывается). Далее выполняется проверка, не равно ли значение элемента $A[q]$ величине x. Если равно, искомый элемент найден, поэтому можно просто вернуть q как индекс элемента массива A, содержащего значение x.

Если же вместо этого выясняется, что $A[q] \neq x$, то можно воспользоваться предположением, что массив A отсортирован. Поскольку $A[q] \neq x$, имеются две возможности: либо $A[q] > x$, либо $A[q] < x$. Сначала рассмотрим случай $A[q] > x$. Так как массив отсортирован, мы знаем, что больше x не только элемент $A[q]$, но и (рассматривая элементы массива как расположенные слева направо) все элементы, расположенные правее $A[q]$. Таким образом, можно исключить из рассмотрения все элементы в позиции q и справа от нее. Поэтому на следующем шаге p не изменяется, а r устанавливается равным $q-1$.



Если же выясняется, что $A[q] < x$, то мы знаем, что каждый элемент массива слева от q меньше, чем x, и поэтому можно эти элементы не рассматривать. Поэтому на следующем шаге r не изменяется, а p устанавливается равным $q+1$.



Вот точное описание процедуры бинарного поиска.

Процедура BINARY-SEARCH(A,n,x)

Вход и выход: те же, что и в LINEAR-SEARCH.

1. Установить p равным 1, а r равным n.
2. Пока $p \le r$, выполнять следующие действия.

   A. Установить q равным $\lfloor (p+r)/2 \rfloor$.

   B. Если $A[q]=x$, вернуть q.

   C. В противном случае ($A[q] \ne x$), если $A[q] > x$, установить r равным $q-1$.

   D. В противном случае ($A[q] < x$) установить r равным $q+1$.
3. Вернуть значение NOT-FOUND.

Цикл на шаге 2 не обязательно завершается из-за того, что p становится больше, чем r. Он может завершиться на шаге 2B, если обнаружит, что $A[q]$ равно x, и вернет q как индекс элемента массива A, равного x.

Для того чтобы показать, что процедура BINARY-SEARCH работает корректно, нам нужно просто показать, что если процедура BINARY-SEARCH возвращает на шаге 3 значение NOT-FOUND, то x отсутствует в массиве. Воспользуемся следующим инвариантом цикла.

В начале каждой итерации цикла в шаге 2, если x находится где-то в массиве A, то это значение находится в одном из элементов подмассива $A[p..r]$.

Вот краткое доказательство корректности с применением инварианта цикла.

Инициализация. Шаг 1 инициализирует индексы p и r значениями 1 и n соответственно, и поэтому инвариант цикла при первом входе в цикл является истинным.

Сохранение. Выше мы доказали, что шаги 2C и 2D корректно обновляют либо p, либо r.

Завершение. Если x отсутствует в массиве, то в конечном итоге процедура доходит до точки, где p и r равны. Когда это происходит, вычисленное на шаге 2A значение q будет таким же, как p и r. Если шаг 2C устанавливает значение r равным $q-1$, то в начале следующей итерации r оказывается равным $p-1$, так что p будет больше r. Если же шаг 2D устанавливает значение p равным $q+1$, то в начале следующей итерации p оказывается равным $r+1$, и p снова будет больше r. В любом случае условие цикла на шаге 2 будет ложным, и цикл завершится. Поскольку $p > r$, подмассив $A[p..r]$ пуст, и, таким образом, значение x не может в нем находиться. Рассматривая контрапозицию инварианта цикла (см. с. 33), находим, что если x отсутствует в подмассиве $A[p..r]$, то его нет нигде в массиве A. Таким образом, процедура корректно возвращает значение NOT-FOUND на шаге 3.

Можно также записать бинарный поиск как рекурсивную процедуру.

Процедура RECURSIVE-BINARY-SEARCH(A,p,r,x)

Вход и выход: входные параметры A и x те же, что и у процедуры LINEAR-SEARCH, также, как и выход. Входные параметры p и r определяют обрабатываемый подмассив $A[p..r]$.

1. Если $p > r$, вернуть NOT-FOUND.
2. В противном случае ($p \le r$) выполнить следующие действия.

   A. Установить $q = \lfloor (p+r)/2 \rfloor$.

   B. Если $A[q] = x$, вернуть q.

   C. В противном случае ($A[q] \ne x$), если $A[q] > x$, вернуть RECURSIVE-BINARY-SEARCH($A,p,q-1,x$).

   D. В противном случае ($A[q] < x$) вернуть RECURSIVE-BINARY-SEARCH($A,q+1,r,x$).

Первоначальный вызов имеет вид RECURSIVE-BINARY-SEARCH($A,1,n,x$).

Теперь давайте рассмотрим, почему бинарный поиск выполняется в массиве с n элементами за время $O(\lg n)$. Ключевым является наблюдение, что размер $r-p+1$ рассматриваемого подмассива уменьшается примерно вдвое на каждой итерации цикла (или при каждом рекурсивном вызове в рекурсивной версии, но давайте пока сосредоточимся на итерационной версии BINARY-SEARCH). Легко видеть, что если итерации начинаются с подмассива из s элементов, то на следующей итерации размер подмассива будет равен либо $\lfloor s/2 \rfloor$, либо $s/2-1$, в зависимости от того, является ли s четным или нечетным числом, и больше или меньше элемент $A[q]$ значения x. Мы уже видели, что, как только размер подмассива опускается до 1, на следующей итерации процедура завершается. Таким образом, вопрос в том, сколько итераций цикла, вдвое уменьшающих рассматриваемый подмассив, нужно выполнить, чтобы его исходный размер n уменьшился до 1. Это то же количество итераций, что и в случае, если бы мы начали с подмассива размером 1 и на каждой итерации удваивали бы его, пока не будет достигнут размер n. Но это просто возведение в степень путем многократного умножения на 2. Другими словами, при каком значении x величина 2^x достигает n? Если n представляет собой точную степень 2, то, как мы уже видели на с. 20, ответом является число $\lg n$. Конечно, n может не быть точной степенью 2, и в этом случае ответ может отличаться от $\lg n$ не более чем на 1. Наконец заметим, что каждая итерация цикла требует постоянного количества времени, т.е. время выполнения отдельной итерации не зависит от размера исходного массива n или от размера рассматриваемого подмассива. Давайте воспользуемся асимптотическими обозначениями для того, чтобы отбросить постоянные множители и члены более низкого порядка. (Равно ли количество итераций $\lg n$ или $\lfloor \lg n \rfloor + 1$? Да какая разница!) В результате мы получаем, что время работы бинарного поиска составляет $O(\lg n)$.

Я использовал здесь O-обозначения, поскольку хотел получить выражение, охватывающее все случаи. В наихудшем случае, когда значение x в массиве отсутствует, мы многократно делим подмассив пополам, пока рассматриваемый подмассив не окажется пустым, и в этом случае время работы равно $\Theta(\lg n)$. В лучшем случае, когда x обнаруживается

в первой же итерации цикла, время работы равно $\Theta(1)$. Нет Θ-обозначения, которое охватывало бы все случаи, но выражение $O(\lg n)$ для времени работы бинарного поиска корректно — конечно, если массив предварительно отсортирован.

Превзойти время $\Theta(\lg n)$ в наихудшем случае можно только при более сложной организации данных и определенных свойствах ключей поиска.

Сортировка выбором

Обратим теперь наше внимание на задачу ***сортировки***, в которой требуется так разместить элементы массива — выполнить их ***перестановку***, — чтобы каждый элемент был меньше или равен следующему за ним. Первый алгоритм сортировки, с которым мы познакомимся, — сортировка выбором, на мой взгляд, является самой простой, потому именно она тут же пришла мне в голову, когда мне надо было разработать алгоритм сортировки массива. Но этот способ сортировки далеко не самый быстрый.

Вот как сортировка выбором будет работать в случае сортировки книг на книжной полке по авторам. Сначала мы проходим по всей полке и находим книгу, автор которой стоит первым по алфавиту, — скажем, Алеся Адамовича (если на полке две или более книг этого автора, выбираем любую из них). Затем мы меняем эту книгу местами с книгой в первом слоте. Теперь в первом слоте находится книга с автором, первым в алфавитном порядке среди всех авторов, книги которых присутствуют на полке. Затем мы вновь проходим по книжной полке слева направо, начиная с книги в слоте 2, и в слотах со второго по слот n ищем книгу, автор которой стоит первым по алфавиту среди просматриваемых книг. Предположим, что это Борис Васильев. Поменяйте эту книгу с книгой в слоте 2, так что теперь слоты 1 и 2 заняты книгами, авторы которых стоят первыми в алфавитном порядке. Затем надо сделать то же самое для слота 3 и т.д. После того как мы поставим нужную книгу в слот $n-1$ (возможно, это книга Шолохова), сортировка выполнена, так как остается только одна книга (скажем, Шукшина), и она находится в том слоте, где и должна находиться, — в слоте n.Чтобы превратить этот рассказ в компьютерный алгоритм, заменим книжную полку массивом, а книги — его элементами. Вот что получается.

Процедура SELECTION-SORT(A,n)

Вход:

- A: сортируемый массив.
- n: количество сортируемых элементов в массиве A.

Результат: элементы массива A отсортированы в неубывающем порядке.

1. Для $i=1$ до $n-1$:

   A. Присвоить переменной *smallest* индекс наименьшего элемента в подмассиве $A[i..n]$.

   B. Обменять $A[i] \leftrightarrow A[smallest]$.

Поиск наименьшего элемента в $A[i..n]$ представляет собой вариант линейного поиска. Сначала объявим наименьшим элементом $A[i]$, а затем просканируем остальную часть подмассива, обновляя индекс наименьшего элемента каждый раз, когда находим элемент, меньший, чем текущий наименьший. Вот какой вид имеет уточненная процедура сортировки.

Процедура Selection-Sort(A,n)

Вход и результат: те же, что и ранее.

1. Для $i = 1$ до $n-1$:

   A. Установить значение переменной *smallest* равным i.

   B. Для $j = i+1$ до n:

      i. Если $A[j] < A[smallest]$, присваиваем переменной *smallest* значение j.

   C. Обменять $A[i] \leftrightarrow A[smallest]$.

Эта процедура имеет так называемый "вложенный" цикл: цикл на шаге 1B вложен в цикл на шаге 1. Внутренний цикл выполняет все свои итерации для каждой отдельной итерации внешнего цикла. Обратите внимание, что начальное значение j во внутреннем цикле зависит от текущего значения i во внешнем цикле. Приведенная ниже иллюстрация показывает, как сортировки выбором работает с массивом из шести элементов.

	1	2	3	4	5	6
A	12	9	3	7	14	11

→

	1	2	3	4	5	6
A	3	9	12	7	14	11

→

	1	2	3	4	5	6
A	3	7	12	9	14	11

→

	1	2	3	4	5	6
A	3	7	9	12	14	11

→

	1	2	3	4	5	6
A	3	7	9	11	14	12

→

	1	2	3	4	5	6
A	3	7	9	11	12	14

Исходный массив показан в верхнем левом углу, и каждое следующее изображение представляет массив после очередной итерации внешнего цикла. Более темным цветом показаны элементы подмассива, о котором точно известно, что он отсортирован.

Если вы хотите воспользоваться инвариантом цикла для доказательства того, что процедура Selection-Sort правильно сортирует массив, вам потребуется по одному инварианту для каждого цикла. Процедура доказательства достаточно проста, так что здесь приведены только инварианты цикла, доказательство остается читателю в качестве небольшого упражнения.

В начале каждой итерации цикла на шаге 1 подмассив $A[1..i-1]$ содержит $i-1$ наименьших элементов массива в отсортированном порядке.

В начале каждой итерации цикла на шаге 1B элемент $A[smallest]$ представляет собой наименьший элемент в подмассиве $A[i..j-1]$.

Чему же равно время работы процедуры Selection-Sort? Покажем, что оно равно $\Theta(n^2)$. Ключевым моментом анализа является выяснение количества итераций, выполняемых внутренним циклом, с учетом того, что каждая итерация выполняется за время $\Theta(1)$. (Константные множители для верхней и нижней границ в Θ-обозначениях могут быть различны, поскольку в каждой конкретной итерации присваивание значения переменной *smallest* может как произойти, так и не произойти.) Давайте посчитаем количество итераций с учетом значения переменной внешнего цикла. Когда i равно 1, внутренний цикл выполняет итерации для j, изменяющегося от 2 до n, или $n-1$ итерацию. Когда i равно 2, внутренний цикл выполняет итерации для j в диапазоне от 3 до n, или $n-2$ итерации. Каждый раз с увеличением переменной внешнего цикла на единицу внутренний цикл выполняется на один раз меньше. В общем случае внутренний цикл выполняется $n-i$ раз. В последней итерации внешнего цикла, когда i равно $n-1$, внутренний цикл выполняется только один раз. Таким образом, общее количество итераций внутреннего цикла равно $(n-1)+(n-2)+(n-3)+\cdots+2+1$.

Такая сумма известна под названием ***арифметическая прогрессия***, и вот очень важный факт об арифметических прогрессиях: для любого неотрицательного целого k

$$k+(k-1)+(k-2)+\cdots+2+1=\frac{k(k+1)}{2}.$$

Подставляя $n-1$ вместо k, мы видим, что общее количество итераций внутреннего цикла равно $(n-1)n/2$, или $(n^2-n)/2$. Воспользуемся асимптотическими обозначениями, чтобы избавиться от членов более низкого порядка ($-n$) и постоянного множителя (1/2). Теперь можно утверждать, что общее количество итераций внутреннего цикла представляет собой $\Theta(n^2)$. Следовательно, время работы сортировки выбором равно $\Theta(n^2)$. Обратите внимание, что это выражение, охватывающее все случаи. Внутренний цикл выполняется $\Theta(n^2)$ раз независимо от фактических значений элементов массива.

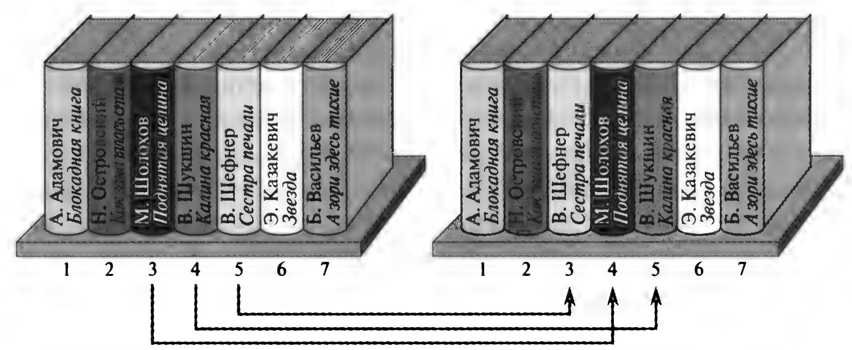
Вот еще один способ увидеть без использования арифметических прогрессий, что время работы равно $\Theta(n^2)$. Покажем отдельно, что время представляет собой как $O(n^2)$, так и $\Omega(n^2)$; объединение асимптотических верхней и нижней границ даст нам $\Theta(n^2)$. Чтобы убедиться, что время работы представляет собой $O(n^2)$, заметим, что в каждой итерации внешнего цикла внутренний цикл выполняется не более $n-1$ раза, что равно $O(n)$, поскольку каждая итерация внутреннего цикла требует постоянного количества времени. Поскольку внешний цикл выполняет итерацию $n-1$ раз, что тоже ни что иное, как $O(n)$, общее время, затраченное во внутреннем цикле, представляет собой $O(n)$, умноженное на $O(n)$, или $O(n^2)$. Чтобы убедиться, что общее время работы представляет собой $\Omega(n^2)$, заметим, что в каждой из первых $n/2$ итераций внешнего цикла внутренний цикл выполняет по меньшей мере $n/2$ итераций, т.е. в общей сложности как минимум $n/2$ умноженное на $n/2$ итераций, или $n^2/4$ итераций. Поскольку каждая итерация внутреннего цикла выполняется за постоянное количество времени, можно сделать вывод, что общее время выполнения процедуры представляет собой по меньшей мере константу, умноженную на $n^2/4$, или $\Omega(n^2)$.

Два последних замечания о сортировке выбором. Во-первых, время работы $\Theta(n^2)$ — наихудшее среди всех алгоритмов сортировки, которые мы будем рассматривать. Во-вторых, если внимательно изучить работу сортировки выбором, мы увидим, что время работы $\Theta(n^2)$ обусловлено сравнениями на шаге 1Bi. Однако количество *перемещений* элементов массива равно только $\Theta(n)$, поскольку шаг 1С выполняется только $n-1$ раз. Если перемещение элементов массива требует большого времени — например, если они велики или располагаются на медленном устройстве типа диска, — то сортировка выбором может оказаться вполне разумным решением.

Сортировка вставкой

Сортировка вставкой несколько отличается от сортировки выбором, хотя и многим на нее похожа. В сортировке выбором при решении, какая книга должна быть положена в слот i, книги в слотах, предшествующих слоту i, уже отсортированы в алфавитном порядке по авторам, причем это первые по алфавиту книги *из всего множества* книг на полке. В случае сортировки вставкой книги в первых i слотах — это те же книги, которые *были изначально в первых i слотах*, но теперь отсортированные в алфавитном порядке по авторам.

В качестве примера давайте предположим, что книги в первых четырех слотах уже отсортированы по автору и что это книги Адамовича, Островского, Шолохова и Шукшина. Пусть книга в слоте 5 написана Шефнером. При сортировке вставкой мы сдвигаем книги Шолохова и Шукшина на один слот вправо, перемещая их из слотов 3 и 4 в слоты 4 и 5, а затем ставим книгу Шефнера в освободившийся слот 3. Пока мы работаем с книгой Шефнера, нас не интересуют книги, стоящие справа от нее (книги Казакевича и Васильева на представленном ниже рисунке), — с ними мы разберемся позже.



Чтобы переместить книги Шолохова и Шукшина, мы сначала сравниваем автора Шукшина с Шефнером. Выяснив, что Шукшин идет после Шефнера, мы сдвигаем его книгу на один слот вправо, из слота 4 в слот 5. Затем сравниваем с Шефнером Шолохова. Выяснив, что Шолохов также идет после Шефнера, мы сдвигаем книгу Шолохова на один слот вправо, из слота 3 в слот 4, который был освобожден при перемещении книги Шукшина. Далее мы сравниваем авторов Шефнера и Островского. На этот раз мы находим, что Островский должен идти перед Шефнером. На этом мы прекращаем сравнения,

так как обнаружили, что книга Шефнера должна быть справа от книги Островского и слева от книги Шолохова. Мы смело можем поставить книгу Шефнера в слот 3, который был освобожден при переносе книги Шолохова.

Переведем описанную идею на язык компьютерного алгоритма для сортировки массива. Подмассив $A[1..i-1]$ содержит только те элементы, которые изначально находились в первых $i-1$ позициях массива, и все они находятся в отсортированном порядке. Чтобы определить, куда надо вставить элемент, первоначально находившийся в $A[i]$, сортировка вставкой проходит по подмассиву $A[1..i-1]$ справа налево, начиная с элемента $A[i-1]$, и переносит каждый элемент, больший, чем $A[i]$, на одну позицию вправо. Как только мы найдем элемент, который не превышает элемент $A[i]$, или доберемся до левого конца массива, мы перенесем элемент, изначально находившийся в $A[i]$, в его новую позицию в массиве.

Процедура INSERTION-SORT(A,n)

Вход и результат: те же, что и в SELECTION-SORT.

1. Для $i = 2$ до n:

   A. Установить переменную key равной $A[i]$, а переменной j присвоить значение $i-1$.

   B. Пока $j > 0$ и $A[j] > key$, выполнять следующее:

      i. Присвоить $A[j+1]$ значение $A[j]$.

      ii. Уменьшить j на единицу (присвоить переменной j значение $j-1$).

   C. Присвоить $A[j+1]$ значение key.

Проверка на шаге 1B использует оператор "и", считая его ***сокращенно вычисляемым***: если выражение слева, $j > 0$, ложно, выражение справа, $A[j] > key$, не вычисляется, поскольку и так очевидно, что общее выражение истинным быть не может. Если этот оператор вычисляет обе части, то при $j \le 0$ обращение к $A[j]$ может привести к сбою выполнения программы.

Вот как сортировка вставками работает с массивом, который уже был использован на с. 44 в качестве примера для сортировки выбором.

	1	2	3	4	5	6
A	12	9	3	7	14	11

→

	1	2	3	4	5	6
A	9	12	3	7	14	11

→

	1	2	3	4	5	6
A	3	9	12	7	14	11

→

	1	2	3	4	5	6
A	3	7	9	12	14	11

→

	1	2	3	4	5	6
A	3	7	9	12	14	11

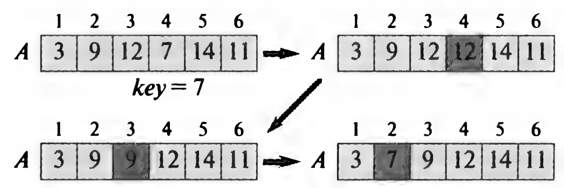
→

	1	2	3	4	5	6
A	3	7	9	11	12	14

Здесь вновь первоначальный массив показан в верхнем левом углу, а каждый шаг показывает, какой вид имеет массив после очередной итерации внешнего цикла на шаге 1. Более темным цветом показаны элементы подмассива, о котором точно известно, что он отсортирован. Инвариант внешнего цикла (который, как и в предыдущем случае, мы приводим без доказательства) имеет следующий вид.

В начале каждой итерации цикла в шаге 1 подмассив $A[1..i-1]$ состоит из элементов, изначально находившихся в $A[1..i-1]$, но теперь — в отсортированном порядке.

На следующем рисунке показано, как внутренний цикл на шаге 1B работает в приведенном выше примере, когда i равно 4. Мы считаем, что подмассив $A[1..3]$ содержит элементы, первоначально располагавшиеся в первых трех позициях массива, но теперь они находятся в отсортированном порядке. Чтобы определить, где следует разместить элемент, первоначально находившийся в позиции $A[4]$, мы сохраняем его в переменной с именем *key*, а затем сдвигаем каждый из элементов $A[1..3]$, больший *key*, на одну позицию вправо.



Выделение темным цветом показывает позиции, в которые перемещаются элементы массива. На последнем показанном шаге значение $A[1]=3$ не превышает значение переменной *key*, равное 7, и внутренний цикл завершается. Как видно из последней части рисунка, значение *key* помещается в позицию, находящуюся непосредственно справа от $A[1]$. Конечно, первоначально требуется сохранить исходное значение $A[i]$ в переменной *key* на шаге 1A, потому что первая итерация внутреннего цикла перезаписывает $A[i]$.

Возможно также, что внутренний цикл завершается из-за невыполнения условия $j > 0$. Эта ситуация возникает, когда значение *key* меньше значений всех элементов в $A[1..i-1]$. Если j становится равным 0, значит, каждый элемент в $A[1..i-1]$ был сдвинут вправо, так что шаг 1C помещает значение *key* в элемент $A[1]$, что и требуется для корректной сортировки элементов.

Анализ времени работы процедуры Insertion-Sort немного сложнее, чем процедуры Selection-Sort. Количество выполнений итераций внутреннего цикла в процедуре Selection-Sort зависит только от индекса i внешнего цикла, но не от самих сортируемых элементов. В случае же процедуры Insertion-Sort количество итераций внутреннего цикла зависит как от индекса i внешнего цикла, так и от значений элементов массива.

Наилучший случай в процедуре Insertion-Sort осуществляется тогда, когда внутренний цикл всякий раз выполняет нуль итераций. Чтобы это произошло, условие $A[j] > key$ должно быть ложным при первой же проверке для каждого значения i. Другими словами, каждый раз, когда выполняется шаг 1B, должно выполняться условие $A[i-1] \le A[i]$. Как это может произойти? Только если массив A отсортирован до выполнения процедуры. В этом случае итерации внешнего цикла выполняются $n-1$ раз, а поскольку каждая итерация внешнего цикла при этом выполняется за постоянное время, процедура Insertion-Sort занимает время $\Theta(n)$.

Наихудший случай осуществляется, когда внутренний цикл каждый раз делает максимально возможное количество итераций. Теперь условие $A[j] > key$ всегда должно быть истинным, и цикл завершается по невыполнению условия $j > 0$. Каждый элемент $A[i]$ должен проделать весь путь до левого конца массива. Как может возникнуть такая ситуация? Только если массив A в начале работы оказывается отсортированным в обратном, *невозрастающем* порядке. В этом случае внешний цикл каждый раз выполняет итерации внутреннего цикла $i-1$ раз. Поскольку переменная внешнего цикла i пробегает все значения от 2 до n, общее количество итераций внутреннего цикла имеет вид арифметической прогрессии

$$1+2+3+\cdots+(n-2)+(n-1),$$

которая, как мы видели при анализе процедуры Selection-Sort, представляет собой $\Theta(n^2)$. Поскольку каждая итерация внутреннего цикла выполняется за константное время, время работы процедуры Selection-Sort в наихудшем случае равно $\Theta(n^2)$. Таким образом, время работы сортировки вставкой в наихудшем случае такое же, как и при сортировке выбором.

Имеет ли смысл попытаться понять, что происходит с сортировкой вставками в среднем? Это зависит от того, как выглядят "средние" входные данные. Если порядок элементов во входном массиве действительно случайный, следует ожидать, что каждый элемент будет больше около половины предшествующих ему элементов и меньше тоже около половины этих элементов, так что каждый раз при выполнении внутреннего цикла последний будет делать примерно $(i-1)/2$ итераций. Это сократит время работы по сравнению с наихудшим случаем в два раза. $1/2$ — это всего лишь постоянный множитель, а потому асимптотически время работы алгоритма остается тем же — $\Theta(n^2)$.

Сортировка вставками является отличным выбором, когда массив изначально "почти отсортирован". Предположим, что каждый элемент массива в начале работы находится в позиции на расстоянии k от той, где оказывается в отсортированном массиве. Тогда общее число перемещений данного элемента в результате всех итераций внутреннего цикла не превышает k. Таким образом, общее количество сдвигов всех элементов в результате всех итераций внутреннего цикла не превышает kn, что, в свою очередь, говорит о том, что общее количество итераций внутреннего цикла также не превышает kn (ведь на каждой итерации внутреннего цикла выполняется сдвиг ровно одного элемента ровно на одну позицию). Если k — константа, то общее время работы сортировки вставками будет составлять $\Theta(n)$, поскольку Θ-обозначение включает постоянный коэффициент k. На самом деле мы даже можем мириться с перемещением некоторых элементов на большие расстояния в массиве, пока таких элементов не слишком много. В частности, если l элементов можно перемещать в массиве произвольным образом (так, что каждый из этих элементов можно переместить на расстояние до $n-1$ позиции), а остальные $n-l$ элементов перемещаются не более чем на k позиций, то общее количество сдвигов не превышает $l(n-1)+(n-l)k=(k+l)n-(k+1)l$, что в случае, когда и l, и k являются константами, составляет $\Theta(n)$.

Если сравнить асимптотическое время сортировки вставкой и выбором, то мы видим, что в наихудшем случае они одинаковы. Сортировка вставками оказывается лучше, если массив почти отсортирован. Сортировка выбором, однако, имеет одно преимущество перед сортировкой вставкой: сортировка выбором перемещает элементы $\Theta(n)$ раз независимо ни от чего, а сортировка вставкой может перемещать элементы до $\Theta(n^2)$ раз, поскольку каждое выполнение шага 1Bi процедуры Insertion-Sort выполняет перемещение элемента. Как мы уже отмечали при рассмотрении сортировки выбором, если перемещение элемента занимает очень много времени и у вас нет оснований ожидать, что сортировка вставкой для конкретных входных данных будет работать лучше, то стоит рассмотреть возможность применения сортировки выбором вместо сортировки вставкой.

Сортировка слиянием

Наш следующий алгоритм сортировки, сортировка слиянием, имеет во всех случаях время работы, равное всего лишь $\Theta(n \lg n)$. Сравнивая его с наихудшим временем работы $\Theta(n^2)$ у алгоритмов сортировки выбором и сортировки вставкой, мы видим, что множитель n заменен множителем $\ln n$ — а такая замена, как мы уже говорили в разделе “Компьютерные алгоритмы для компьютерщиков” на с. 19, однозначно выгодна.

Однако сортировка слиянием имеет и пару недостатков по сравнению с уже рассмотренными нами алгоритмами сортировки. Во-первых, постоянный множитель, прячущийся в асимптотических обозначениях, оказывается бóльшим, чем у других двух алгоритмов. Конечно, когда размер массива n становится достаточно большим, это перестает иметь значение. Во-вторых, сортировка слиянием не работает “***на месте***”, без привлечения дополнительной памяти: она делает полные копии всего входного массива. Сравните эту функцию с сортировкой выбором и сортировкой вставкой, которые в любой момент хранят дополнительные копии только одной записи массива, а не всего массива целиком. Если вопрос использования памяти приоритетен, использовать сортировку слиянием нельзя.

В алгоритме используется распространенная алгоритмическая парадигма, известная как “***разделяй и властвуй***”. В ней мы разделяем задачу на подзадачи, которые подобны исходной, решаем подзадачи рекурсивно, а затем объединяем решения подзадач в решение исходной задачи. Вспомним из главы 2, “Описание и оценка компьютерных алгоритмов”, что, чтобы рекурсия работала, каждый рекурсивный вызов должен работать с меньшим экземпляром той же задачи, чтобы в конечном итоге достичь базового случая. Вот общая схема работы алгоритма “разделяй и властвуй”.

1. ***Разделение.*** Задача разбивается на несколько подзадач, которые представляют собой меньшие экземпляры той же самой задачи.

2. ***Властвование.*** Рекурсивно решаются подзадачи. Если они достаточно малы, они решаются как базовый случай.

3. ***Объединение.*** Решения подзадач объединяются в решение исходной задачи.

При сортировке книг на книжной полке с помощью сортировки слиянием каждая подзадача состоит из сортировки книг в последовательных слотах полки. Первоначально мы хотим отсортировать все n книг в слотах от первого до n-го, но подзадача в общем случае состоит в упорядочении всех книг в слотах от p-го до r-го. Вот как применяется парадигма "разделяй и властвуй" в этом случае.

1. ***Разделение.*** Разделяем сортируемый промежуток путем нахождения значения q посредине между p и r. Мы делаем это так же, как и при бинарном поиске: $q = \lfloor (p+r)/2 \rfloor$.
2. ***Властвование.*** Рекурсивно сортируем книги в каждой половине промежутка, созданной на шаге разделения: рекурсивно сортируем промежуток слотов от p до q, и от $q+1$ до r.
3. ***Объединение.*** Объединение отсортированных книг в промежутках слотов от p до q и от $q+1$ до r так, чтобы книги в промежутке от p-го до r-го слотов были отсортированы. Как это сделать, мы вскоре узнаем.

Базовый случай осуществляется, когда надо сортировать менее двух книг (т.е. когда $p \geq r$), поскольку множество книг, в котором книг нет или имеется единственная книга, уже отсортировано тривиальным образом.

Чтобы превратить эту идею в алгоритм сортировки массива, книги в слотах от p до r рассматриваются как подмассив $A[p..r]$. В алгоритме используется процедура MERGE(A,p,q,r), которая сливает отсортированные подмассивы $A[p..q]$ и $A[q+1..r]$ в единый отсортированный подмассив $A[p..q]$.

Процедура MERGE-SORT(A,p,r)

Вход:

- A: массив
- p, r: начальный и конечный индексы подмассива A

Результат: элементы подмассива $A[p..r]$ отсортированы в неубывающем порядке.

1. Если $p \geq r$, подмассив $A[p..r]$ содержит не более одного элемента, так что он автоматически является отсортированным. Выполняем возврат из процедуры без каких-либо действий.
2. В противном случае выполняем следующие действия.

   A. Установить $q = \lfloor (p+r)/2 \rfloor$.

   B. Рекурсивно вызвать MERGE-SORT(A,p,q)

   C. Рекурсивно вызвать MERGE-SORT($A,q+1,r$)

   D. Вызвать MERGE(A,p,q,r)

Хотя мы пока что не видели, как работает процедура Merge, уже можно посмотреть, как работает процедура Merge-Sort. Начнем со следующего массива.

1	2	3	4	5	6	7	8	9	10
12	9	3	7	14	11	6	2	10	5

Первоначальный вызов для сортировки всего массива — Merge-Sort(A,1,10). На шаге 2A находим, что q равно 5, так что рекурсивными вызовами на шагах 2B и 2C являются Merge-Sort(A,1,5) и Merge-Sort(A,6,10).

1	2	3	4	5
12	9	3	7	14

6	7	8	9	10
11	6	2	10	5

После возврата из этих двух рекурсивных вызовов подмассивы будут отсортированы.

1	2	3	4	5
3	7	9	12	14

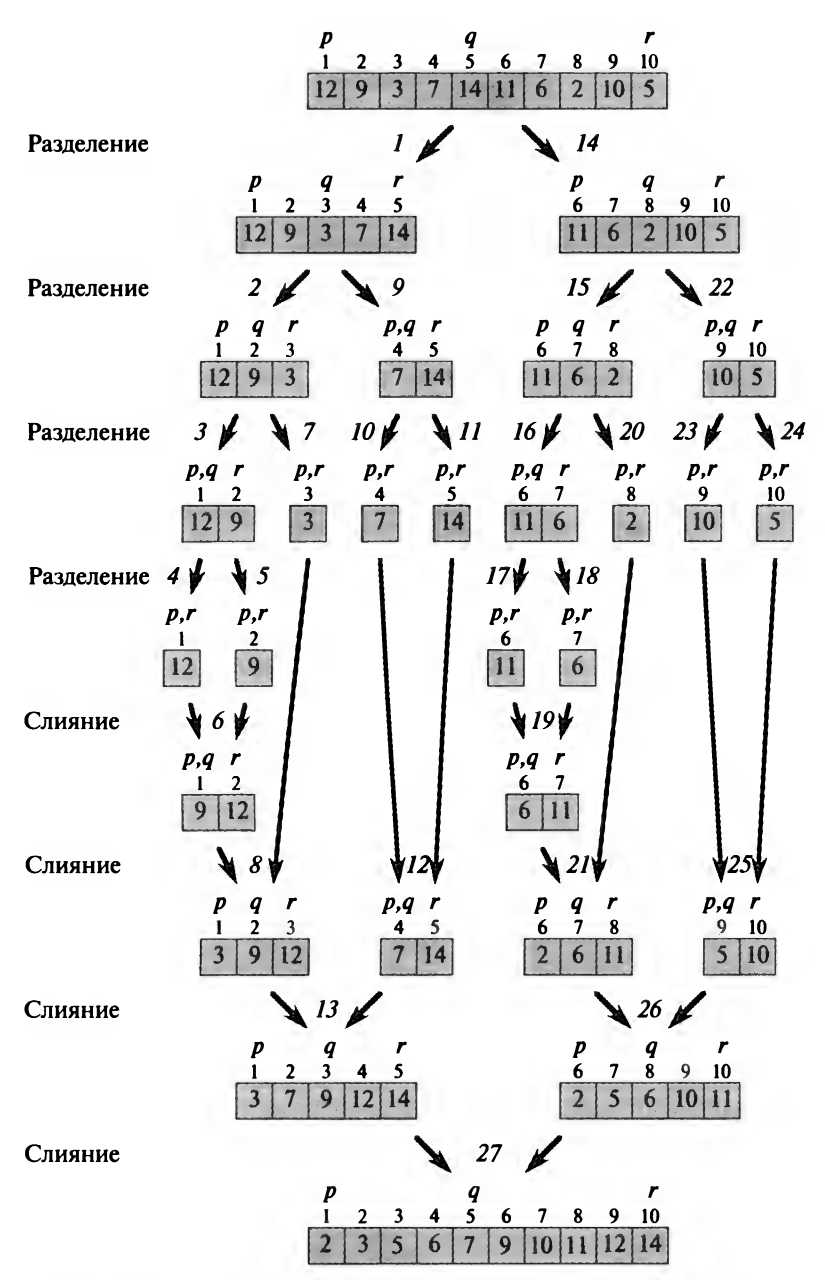
6	7	8	9	10
2	5	6	10	11

Наконец вызов Merge(A,1,5,10) на шаге 2D сливает два отсортированных подмассива в единый отсортированный подмассив, который в данном случае представляет собой весь массив полностью.

1	2	3	4	5	6	7	8	9	10
2	3	5	6	7	9	10	11	12	14

Если полностью раскрыть рекурсивные вызовы, мы получим приведенную далее схему. Расходящиеся стрелки показывают шаги разделения, а сходящиеся — шаги слияния. Переменные p, q и r, показанные над каждым подмассивом, располагаются над индексами, которым они соответствуют в каждом рекурсивном вызове. Выделенные курсивом числа указывают порядок, в котором осуществляются вызовы процедур после первоначального вызова Merge-Sort(A,1,10). Например, вызов Merge(A,1,3,5) является тринадцатым вызовом после начального, а вызов Merge-Sort(A,6,7) — шестнадцатым.

Реальная работа происходит в процедуре Merge. Таким образом, эта процедура должна работать не только корректно, но еще и быстро. Если сливаются в общей сложности n элементов, то лучшее, на что мы можем надеяться, — это время работы $\Theta(n)$, поскольку каждый элемент должен быть поставлен в соответствующее место. И мы действительно можем достичь слияния за линейное время.



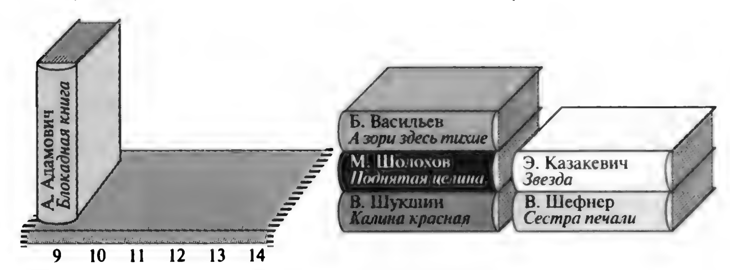
Возвращаясь к примеру с книжной полкой, рассмотрим часть полки со слотами с 9 по 14. Предположим, что у нас имеются отсортированные книги в слотах 9–11 и в слотах 12–14.



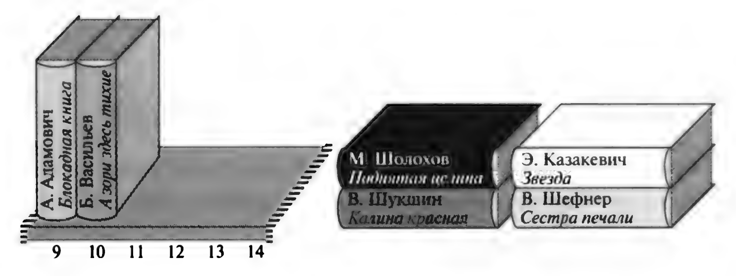
Снимем с полки книги в слотах 9–11 и положим их стопкой, так что книга автора, идущего первым по алфавиту, оказывается на вершине стопки. Точно так же сложим книги из слотов 12–14, сделав вторую стопку.



Поскольку эти стопки уже отсортированы, книга в слоте 9 должна быть одной из верхних книг этих двух стопок: либо книга Васильева, либо книга Адамовича. Мы видим, что книга Адамовича должна находиться до книги Васильева, так что мы ставим ее в слот 9.

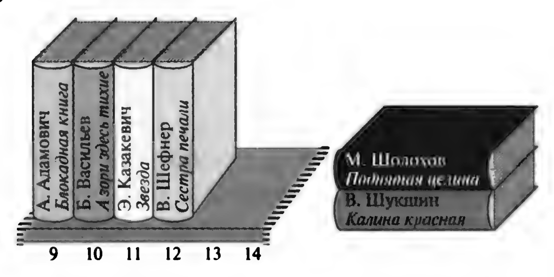


После того как книга Адамовича помещена в слот 9, книга, которая должна быть поставлена в слот 10, должна быть верхней книгой одной из стопок — либо книгой Васильева, либо книгой Казакевича. Поэтому в слот 10 мы ставим книгу Васильева.



Затем мы вновь сравниваем книги на вершинах стопок — теперь это книги Шолохова и Казакевича — и ставим в слот 11 книгу Казакевича. В правой стопке остается одна книга

Шефнера, и после сравнения с Шолоховым мы ставим ее в слот 12. В этот момент правая стопка становится пустой.



Все, что нам осталось сделать, — это взять книги из левой стопки и поставить их в оставшиеся пустыми слоты в том порядке, в котором они лежат в стопке.



Насколько эффективна эта процедура слияния? Каждую книгу мы перемещаем ровно два раза: снимая ее с книжной полки и вновь возвращая на полку. Кроме того, когда мы решаем, какая из книг должна быть поставлена на полку, нам надо сравнивать только две книги, находящиеся на вершинах стопок. Следовательно, чтобы выполнить слияние n книг, нам надо переместить книги $2n$ раз и сравнить пары книг не более n раз.

Зачем мы снимаем книги с полки? Что произойдет, если мы оставим книги на полке и просто будем отслеживать, какие книги мы уже поместили в их корректные места на полке, а какие еще нет? Это может привести к намного большему количеству работы. Предположим, например, что каждая книга в правой половине полки должна в конечном итоге оказаться перед каждой из книг в левой половине. Прежде чем мы смогли бы переместить первую книгу из правой половины в первый слот левой половины, нам бы потребовалось перенести каждую из книг в левой половине вправо на один слот для того, чтобы освободить место для переносимой книги. Затем нам пришлось бы делать то же самое, чтобы поставить следующую книгу, и так далее для всех прочих книг, которые изначально находились в правой половине полки. В результате при установке на место каждой книги из правой половины полки нам пришлось бы перемещать половину всех книг на полке.

Эти рассуждения поясняют нам, почему мы не можем выполнять слияние "на месте", без привлечения дополнительной памяти.[2] Возвращаясь к слиянию отсортированных подмассивов $A[p..q]$ и $A[q+1..r]$ в подмассив $A[p..r]$, мы начинаем с копирования сливаемых элементов из массива A во временные массивы, а затем возвращаем их в массив A.

[2] *На самом деле слияние на месте за линейное время выполнить можно, но соответствующая процедура оказывается неоправданно сложной.*

Пусть $n_1 = q - p + 1$ — количество элементов в $A[p..q]$, а $n_2 = r - q$ — количество элементов в $A[q+1..r]$. Создадим временные массивы B с n_1 элементами и C с n_2 элементами и скопируем элементы из $A[p..q]$, не нарушая их порядок, в массив B, а элементы из $A[q+1..r]$ — точно так же в массив C. Теперь можно вернуть эти элементы назад в $A[p..r]$, не боясь перезаписать имеющиеся там элементы копиями.

Мы сливаем элементы массива точно так же, как книги. Будем копировать элементы массивов B и C обратно в подмассив $A[p..r]$, поддерживая индексы в этих массивах для того, чтобы отслеживать наименьшие элементы, все еще не скопированные из массивов B и C, и копировать обратно минимальный из них. За константное время мы можем выяснить, какой из элементов меньше, скопировать его в корректную позицию в $A[p..r]$ и обновить индексы в массивах.

В конце концов один из двух массивов будет полностью перенесен обратно в $A[p..r]$. Эта ситуация соответствует моменту, когда остается только одна стопка книг. Чтобы избежать необходимости проверять каждый раз, не исчерпался ли полностью один из массивов, прибегнем к хитрости, разместив в правом конце каждого из массивов B и C дополнительный элемент, который заведомо больше любого другого элемента. Помните трюк с ограничителем, который мы использовали в процедуре SENTINEL-LINEAR-SEARCH в главе 2, "Описание и оценка компьютерных алгоритмов"? Используемая при слиянии идея очень похожа на трюк с ограничителем. Здесь мы используем в качестве ключа ограничителя значение ∞ (бесконечность)[3], так что, когда все элементы из обоих массивов, B и C, скопированы обратно в исходный массив, в обоих массивах в качестве наименьших элементов остаются ограничители. Но сравнивать при этом ограничители между собой не потребуется, потому что к этому времени все "реальные" элементы уже скопированы обратно в $A[p..r]$. Поскольку мы заранее знаем, что будем копировать назад элементы начиная с $A[p]$ и заканчивая $A[r]$, мы можем прекратить копирование, как только скопируем элемент $A[r]$. Таким образом, можно просто использовать цикл, в котором индекс в массиве A пробегает значения от p до r.

Вот как выглядит процедура MERGE. Она кажется длинной, но она всего лишь строго следует описанному выше методу.

Процедура MERGE(A,p,q,r)

Вход:

- A: массив.
- p, q, r: индексы в массиве A. Подмассивы $A[p..q]$ и $A[q+1..r]$ считаются уже отсортированными.

Результат: подмассив $A[p..r]$, содержащий все элементы, изначально находившиеся в подмассивах $A[p..q]$ и $A[q+1..r]$, но теперь подмассив $A[p..r]$ отсортирован.

1. Установить n_1 равным $q - p + 1$, а n_2 — равным $r - q$.

[3] *На практике представляется значением, заведомо большим любого ключа сортировки. Например, при сортировке по авторам это может быть "Яяяяя" — конечно, в предположении, что такого автора на самом деле не существует.*

2. $B[1..n_1+1]$ и $C[1..n_2+1]$ представляют собой новые массивы.
3. Скопировать $A[p..q]$ в $B[1..n_1]$, а $A[q+1..r]$ — в $C[1..n_2]$.
4. Установить $B[n_1+1]$ и $C[n_2+1]$ равными ∞.
5. Установить i и j равными 1.
6. Для $k = p$ до r

   A. Если $B[i] \le C[j]$, установить $A[k]$ равным $B[i]$ и увеличить i на 1.

   B. В противном случае ($B[i] > C[j]$) установить $A[k]$ равным $C[j]$ и увеличить j на 1.

В результате шагов 1–4 выделяется память для массивов B и C, выполняется копирование $A[p..q]$ в B и $A[q+1..r]$ в C, а также вставка в эти массивы ограничителей. После этого каждая итерация цикла на шаге 6 копирует наименьший из оставшихся в массивах B и C элементов в очередную позицию $A[p..r]$, и цикл завершается после того, как обратно в $A[p..r]$ будут скопированы все элементы из массивов B и C. В этом цикле i индексирует наименьший остающийся элемент в B, а j — наименьший остающийся элемент в C. Индекс k указывает позицию в A, куда будет помещен очередной копируемый элемент.

Если мы сливаем вместе n элементов (так что $n = n_1 + n_2$), этот процесс требует времени $\Theta(n)$ для копирования элементов в массивы B и C и константного времени для копирования каждого элемента из массивов B и C обратно в A, так что суммарное время работы алгоритма равно $\Theta(n)$.

Ранее мы утверждали, что алгоритм сортировки слиянием имеет время работы $\Theta(n \lg n)$. Мы сделаем упрощающее предположение о том, что размер массива n представляет собой степень 2, так что каждый раз, когда мы делим массив пополам, размеры подмассивов равны. (В общем случае n может не быть степенью 2, а потому размеры подмассивов могут не быть равными в конкретном рекурсивном вызове. Справиться с этой ситуацией может строгий математический анализ, но в этой книге мы не будем утруждать себя его проведением.)

Вот как мы анализируем время работы сортировки слиянием. Пусть сортировка подмассива из n элементов занимает время $T(n)$, которое представляет собой функцию, растущую с ростом n (очевидно, что чем больше элементов, тем больше требуется времени для их сортировки). Время $T(n)$ состоит из трех суммируемых воедино компонентов парадигмы “разделяй и властвуй”.

1. Разделение занимает константное время, поскольку состоит только в вычислении индекса q.
2. Властвование состоит из двух рекурсивных вызовов для подмассивов, каждый размером $n/2$ элементов. В соответствии с определением времени сортировки подмассива каждый из этих двух рекурсивных вызовов занимает время $T(n/2)$.
3. Объединение результатов двух рекурсивных вызовов с помощью слияния отсортированных подмассивов выполняется за время $\Theta(n)$.

Поскольку константное время для разделения имеет более низкий по сравнению со временем объединения $\Theta(n)$ порядок, оно поглощается временем слияния, и можно считать, что разделение и объединение вместе выполняются за время $\Theta(n)$. Шаг властвования выполняется за время $T(n/2)+T(n/2)$, или $2T(n/2)$. Таким образом, мы можем записать уравнение для $T(n)$ следующим образом:

$$T(n)=2T(n/2)+f(n),$$

где $f(n)$ представляет время, необходимое для разделения и слияния, которое, как мы уже упоминали, представляет собой $\Theta(n)$. В изучении алгоритмов распространенной практикой является помещение в правой части уравнения асимптотических обозначений для представления некоторой функции, не заботясь о том, чтобы присвоить ей имя, так что указанное уравнение можно переписать как
$T(n)=2T(n/2)+\Theta(n)$.

Минутку! Похоже, здесь что-то не так. Мы определили функцию T, которая описывает время работы сортировки слиянием, через ту же самую функцию! Мы называем такую запись ***рекуррентным уравнением***, или просто ***рекуррентностью***. Проблема заключается в том, что мы хотим выразить $T(n)$ нерекурсивным способом, т.е. не через саму эту функцию. Такая задача может оказаться настоящей головной болью, но для широкого класса рекуррентных уравнений можно применить метод, известный как "основной метод". Он применим ко многим (хотя и не ко всем) рекуррентностям вида $T(n)=aT(n/b)+f(n)$, где a и b — положительные целые константы. К счастью, он применим и к нашему рекуррентному соотношению для сортировки слиянием и дает результат, состоящий в том, что $T(n)$ имеет вид $\Theta(n\lg n)$.

Это время — $\Theta(n\lg n)$ — относится ко всем случаям сортировки слиянием — наилучшему случаю, наихудшему случаю и ко всем промежуточным. Каждый элемент копируется $\Theta(n\lg n)$ раз. Как видно из рассмотрения метода MERGE, когда он вызывается с $p=1$ и $r=n$, он делает копии всех n элементов, так что сортировка слиянием, определенно, не работает "на месте".

Быстрая сортировка

Как и сортировка слиянием, быстрая сортировка использует парадигму "разделяй и властвуй" (а следовательно, использует рекурсию). Однако применение принципа "разделяй и властвуй" быстрой сортировкой несколько отличается от случая сортировки слиянием. Быстрая сортировка имеет и пару других существенных отличий от сортировки слиянием.

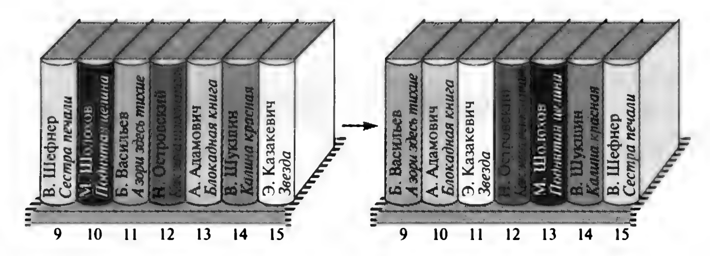
- Быстрая сортировка работает "на месте", без привлечения дополнительной памяти.
- Асимптотическое время работы быстрой сортировки для среднего случая отличается от времени работы для наихудшего случая.

У быстрой сортировки, кроме того, достаточно хороший постоянный множитель (лучше, чем у сортировки слиянием), так что на практике чаще всего предпочтение отдается быстрой сортировке.

Вот как быстрая сортировка применяет парадигму “разделяй и властвуй”. Вновь вернемся к нашей книжной полке. Как и в случае сортировки слиянием, изначально мы хотим отсортировать все n книг в слотах с первого до n-го и при этом рассматриваем обобщенную задачу сортировки книг в слотах с p по r.

1. ***Разделение.*** Сначала выберем одну книгу из слотов от p по r. Назовем эту книгу ***опорной***. Переставим книги на полке так, чтобы все книги с авторами, идущими в алфавитном порядке до автора опорной книги (или с автором, совпадающим с автором опорной книги), находились слева от опорной, а книги с авторами, идущими по алфавиту после автора опорной книги, — справа от последней.

   В данном примере выберем крайнюю справа книгу — Казакевича — в качестве опорной для перестановки книг в слотах с 9 по 15.

   После перестановки — которая в быстрой сортировке носит название ***разбиения*** — книги Васильева и Адамовича, идущих по алфавиту до Казакевича, оказываются слева от книги Казакевича, а все остальные книги, авторы которых в алфавитном порядке идут после Казакевича, оказываются справа. Заметим, что после разбиения книги как слева от книги Казакевича, так и справа, не располагаются в каком-то конкретном порядке.

2. ***Властвование.*** Осуществляется путем рекурсивной сортировки книг слева и справа от опорного элемента. То есть если при разделении опорный элемент вносится в слот q (слот 11 в нашем примере), то рекурсивно сортируются книги в слотах с p по $q-1$ и с $q+1$ по r.

3. ***Объединение.*** На этом этапе мы ничего не делаем! После рекурсивной сортировки мы получаем полностью отсортированный массив. Почему? Авторы всех книг слева от опорной (в слотах с p по $q-1$) идут по алфавиту до автора опорной книги, и книги отсортированы, а авторы всех книг справа от опорной (в слотах с $q+1$ по r) идут по алфавиту после автора опорной книги, и все эти книги также отсортированы. То есть отсортированы все книги в слотах начиная со слота p и заканчивая слотом r!

При замене книжной полки массивом, а книг — элементами массива мы получим компьютерный алгоритм быстрой сортировки. Подобно сортировке слиянием, базовый случай осуществляется, когда сортируемый подмассив содержит менее двух элементов.

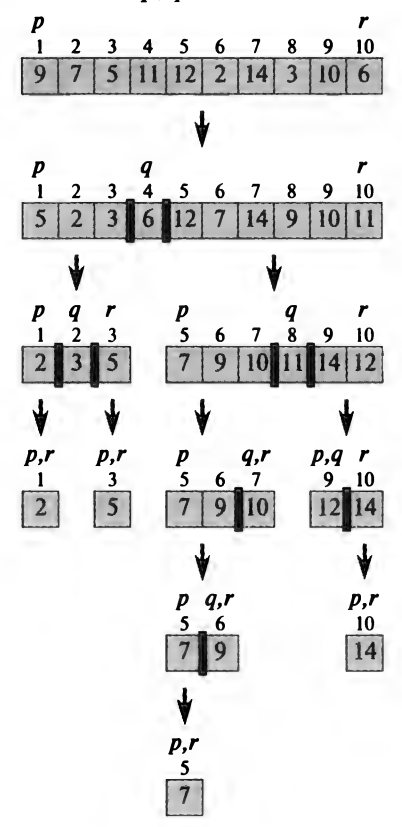
Процедура быстрой сортировки подразумевает вызов процедуры PARTITION(A,p,r), которая разбивает подмассив $A[p..r]$ и возвращает индекс q позиции, в которую помещается опорный элемент.

Процедура QUICKSORT(A,p,r)

Вход и результат: те же, что и у процедуры MERGE-SORT.

1. Если $p \geq r$, просто выйти из процедуры, не выполняя никаких действий.
2. В противном случае выполнить следующее.
   A. Вызвать PARTITION(A,p,r) и установить значение q равным результату вызова.
   B. Рекурсивно вызвать QUICKSORT($A,p,q-1$).
   C. Рекурсивно вызвать QUICKSORT($A,q+1,r$).

Первоначальный вызов QUICKSORT($A,1,n$) аналогичен вызову процедуры MERGE-SORT. Вот пример работы процедуры QUICKSORT с развернутой рекурсией. Для каждого подмассива, в котором $p \leq r$, указаны индексы p, q и r.



Самое нижнее значение в каждой позиции массива показывает, какой элемент будет находиться в этой позиции по завершении сортировки. При чтении массива слева направо смотрите на самые нижние значения в каждой позиции, и вы убедитесь в том, что массив отсортирован.

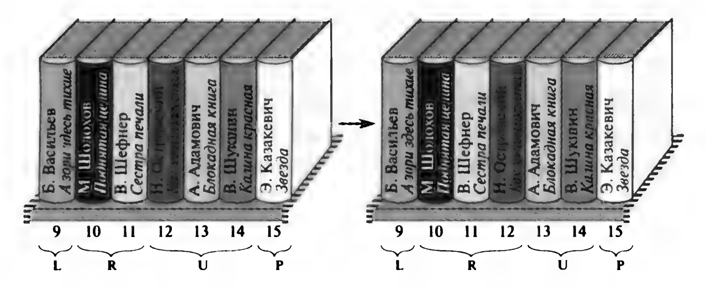
Ключом к быстрой сортировке является разбиение. Так же, как мы смогли слить n элементов за время $\Theta(n)$ в сортировке слиянием, мы можем разбить n элементов за время $\Theta(n)$.

Вот как мы будем разбивать книги, которые находятся на полке в слотах с p по r. В качестве опорной выбираем крайнюю справа книгу — из слота r. В любой момент каждая книга может находиться в одной из четырех групп, и эти группы располагаются в слотах от p до r слева направо.

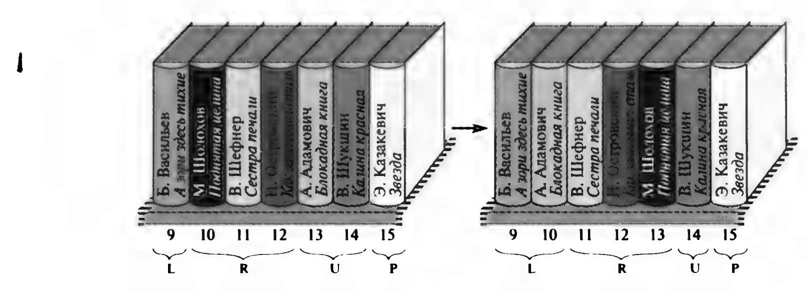
- Группа L (левая): книги с авторами, о которых известно, что они располагаются в алфавитном порядке до автора опорной книги или написаны автором опорной книги.
- Далее идет группа R (правая): книги с авторами, о которых известно, что они располагаются в алфавитном порядке после автора опорной книги.
- Затем идет группа U (неизвестная): книги, которые мы еще не рассмотрели и не знаем, как их авторы располагаются по отношению к автору опорной книги в алфавитном порядке.
- Последней идет группа P (опорная): в нее входит единственная опорная книга.

Мы проходим по книгам группы U слева направо, сравнивая каждую из них с опорной и перемещая ее либо в группу L, либо в группу R, останавливаясь по достижении опорной книги. Книга, которую мы сравниваем с опорной, — всегда крайняя слева в группе U.

- Если автор книги находится в алфавитном порядке после автора опорной книги, то книга становится крайней справа в группе R. Поскольку до этого она была крайней слева в группе U, а за группой U непосредственно следует группа R, мы должны просто переместить разделительную линию между группами R и U на один слот вправо, без перемещения каких-либо книг.



- Если автор книги находится в алфавитном порядке до автора опорной книги или совпадает с автором опорной книги, то эта книга становится крайней справа в группе L. Мы обмениваем ее с крайней слева книгой в группе R и перемещаем разделительную линию между группами L и R и между группами R и U на один слот вправо.



Добравшись до опорной книги, мы обмениваем ее с крайней слева книгой группы R. В нашем примере разбиение завершается расстановкой книг, показанной на рис. на с. 59.

Мы сравниваем каждую книгу с опорной один раз, и с каждой книгой, автор которой находится в алфавитном порядке до автора опорной книги или совпадает с автором опорной книги, выполняется один обмен. Для разбиения n книг, таким образом, делается не более $n-1$ сравнения (так как нам не нужно сравнить опорную книгу саму с собой) и не более n обменов. Обратите внимание, что, в отличие от слияния, книги можно разбить без снятия их всех с полки, т.е. разбиение выполняется на месте, без привлечения дополнительной памяти.

Чтобы преобразовать разбиение книг на полке в разбиение подмассива $A[p..r]$, мы сначала выбираем крайний справа элемент $A[r]$ в качестве опорного. Затем мы проходим через подмассив слева направо, сравнивая каждый элемент с опорным. Мы поддерживаем в подмассиве индексы q и u, которые разделяют его следующим образом:

- подмассив $A[p..q-1]$ соответствует группе L: каждый его элемент не превышает опорный;
- подмассив $A[q..u-1]$ соответствует группе R: каждый его элемент больше опорного;
- подмассив $A[u..r-1]$ соответствует группе U: нам пока неизвестно, как его элементы соотносятся с опорным;
- элемент $A[r]$ соответствует группе P: это опорный элемент.

Это разделение, по сути, представляет собой инвариант цикла (но мы не будем его доказывать).

На каждом шаге мы сравниваем крайний слева элемент группы U $A[u]$ с опорным элементом. Если $A[u]$ больше опорного элемента, мы увеличиваем u, чтобы переместить разделительную линию между группами R и U вправо. Если же $A[u]$ не превышает опорный элемент, мы обмениваем элементы $A[q]$ (крайний слева элемент в группе R) и $A[u]$, а затем увеличиваем q и перемещаем разделительные линии между группами L и R и группами R и U вправо. Вот как выглядит процедура Partition.

Процедура Partition(A,p,r)

Вход: тот же, что и для Merge-Sort.

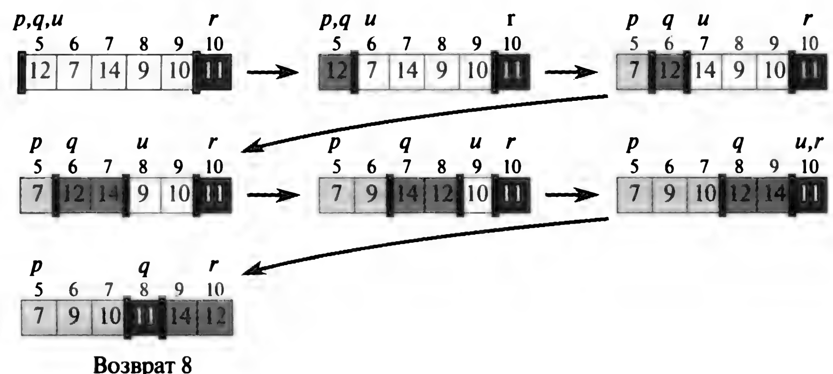
Результат: перестановка элементов $A[p..r]$, такая, что каждый элемент в $A[p..q-1]$ не превышает $A[q]$, а каждый элемент в $A[q+1..r]$ больше $A[q]$. Возвращает значение индекса q.

1. Установить q равным p.
2. Для $u = p$ до $r-1$:

   A. Если $A[u] \le A[r]$, обменять $A[q]$ с $A[u]$, а затем увеличить q на 1.
3. Обменять $A[q]$ и $A[r]$, а затем вернуть q.

Поскольку изначально значения обоих индексов q и u равны p, группы L ($A[p..q-1]$) и R ($A[q..u-1]$) в начале работы алгоритма пустые, а группа U ($A[u..r-1]$) содержит при этом все элементы, за исключением опорного. В некоторых случаях, например, при $A[p] \leq A[r]$, элемент может меняться местами с самим собой, что не влечет за собой никаких изменений в массиве. Шаг 3 заканчивается обменом опорного элемента с крайним слева элементом в группе R, тем самым опорный элемент перемещается в его правильное место в разбитом массиве, после чего процедура возвращает новый индекс q опорного элемента.

Вот как пошагово работает процедура Partition с подмассивом $A[5..10]$, созданным при первом разбиении в примере быстрой сортировки на с. 62. Группа U показана белым цветом, группа L имеет легкое затенение, группа R окрашена в более темный цвет, и темнее всех изображен опорный элемент, который представляет собой группу P. В первой части рисунка показаны исходные массив и индексы, следующие пять частей показывают массив и индексы после каждой итерации цикла на шаге 2 (включая увеличение индекса u в конце каждой итерации), а последняя часть показывает окончательный разбитый массив.



Как и при разбиении множества книг на книжной полке, мы по одному разу сравниваем каждый элемент с опорным и выполняем не более одного обмена для каждого элемента, который сравниваем с опорным. Поскольку и каждое сравнение, и каждый обмен занимают константное время, общее время работы процедуры Partition с n-элементным подмассивом равно $\Theta(n)$.

Так каково же время работы процедуры Quicksort? Обозначим, как и в случае сортировки слиянием, время сортировки подмассива из n элементов как $T(n)$ — функцию, которая увеличивается с ростом n. Разбиение с помощью процедуры Partition занимает время $\Theta(n)$. Но время работы быстрой сортировки зависит от того, как именно выполняется разбиение.

В наихудшем случае размеры разделов являются несбалансированными. Если каждый элемент, отличный от опорного, оказывается меньше последнего, разбиение оставляет опорный элемент в $A[r]$ и возвращает индекс r процедуре Quicksort, которая сохраняет это значение в переменной q. В этом случае подмассив $A[q+1..r]$ является пустым, а массив $A[p..q-1]$ только на один элемент меньше, чем массив $A[p..r]$. Рекурсивный вызов для пустого подмассива выполняется за время $\Theta(1)$ (время, необходимое для осуществления

вызова и определения на шаге 1, что подмассив пуст). Этим временем можно пренебречь и скрыть его во времени $\Theta(n)$, необходимом для разбиения. Но если $A[p..r]$ имеет n элементов, то $A[p..q-1]$ содержит $n-1$ элемент, а потому рекурсивный вызов для подмассива $A[p..q-1]$ занимает время $T(n-1)$. Таким образом, мы получаем рекуррентное соотношение $T(n)=T(n-1)+\Theta(n)$.

Хотя мы и не можем решить это рекуррентное соотношение с помощью основного метода, оказывается, что $T(n)$ имеет вид $\Theta(n^2)$. Это не лучше, чем сортировка выбором! Но как мы можем получить такое неравномерное разбиение? Только если каждый опорный элемент больше всех прочих элементов, т.е. массив должен быть изначально отсортирован. Оказывается также, что мы получаем неравномерное разделение и тогда, когда массив изначально отсортирован в обратном порядке.

С другой стороны, если всякий раз каждый из подмассивов будет иметь размер $n/2$, то рекуррентное соотношение для времени работы окажется таким же, как и рекуррентное соотношение на с. 58 для сортировки слиянием, а именно

$$T(n)=2T(n/2)+\Theta(n),$$

и будет иметь то же самое решение — что $T(n)$ представляет собой $\Theta(n\lg n)$. Конечно, надо быть удивительным везунчиком, чтобы всякий раз при разбиении подмассива он разбивался на строго равные части.

Обычный случай лежит где-то посредине между наилучшим и наихудшим. Математический анализ этого вопроса достаточно сложен, и я не хочу мучить вас им, так что изложу только выводы: если элементы входного массива располагаются в случайном порядке, то в среднем мы получаем разделения, достаточно близкие к разбиениям пополам, так что быстрая сортировка имеет при этом время работы $\Theta(n\lg n)$.

Теперь давайте ненадолго станем параноиками. Предположим, что ваш злейший враг дал вам массив для сортировки, зная, что вы всегда выбираете в качестве опорного последний элемент в каждом подмассиве, и организовал массив так, что вы всегда будете получать наихудшие разбиения. Как помешать этому злокозненному врагу? Конечно, можно начать с проверки, не отсортирован ли массив изначально, и поступать в таких случаях особым образом. Но ведь ваш враг тоже не дурак и может придумать массив, в котором разбиения всегда плохие, но не предельно плохие. Вы же не будете проверять все возможные плохие случаи?

К счастью, есть гораздо более простое решение: не всегда выбирать в качестве опорного последний элемент. Но ведь тогда тщательно выверенная процедура Partition не будет работать, потому что группы элементов окажутся не на своих местах? Это не проблема: достаточно перед выполнением процедуры Partition поменять $A[r]$ с некоторым произвольно выбранным элементом из $A[p..r]$. Теперь опорный элемент выбран случайным образом, так что далее вы можете смело запускать обычную процедуру Partition.

В действительности ценой небольших усилий можно повысить шансы на получение хороших разбиений. Вместо случайного выбора одного элемента из $A[p..r]$ выберите три случайных элемента и обменяйте с $A[r]$ их медиану. Под медианой трех элементов подразумевается тот элемент, значение которого находится между двумя другими (если два или

более из случайно выбранных элементов равны, выберите медиану произвольно). Я вновь не хочу мучить вас анализом, но вам нужно быть действительно уникально невезучим, чтобы процедура Quicksort при этом имела время работы большее, чем $\Theta(n \lg n)$. Кроме того, если только ваш враг не имеет доступа к вашему генератору случайных чисел, он никоим образом не сможет устроить вам козни и затормозить работу сортировки путем подбора соответствующих входных данных.

Сколько же раз процедура Quicksort обменивает элементы? Это зависит от того, считать ли "обменом" ситуацию, когда элемент должен обменяться местами с самим собой. Естественно, всегда можно проверить, не пытаемся ли мы обменивать элемент сам с собой, и, если это так, не выполнять никаких действий. Поэтому будем считать обменом только те случаи, когда элементы действительно перемещаются в массиве, т.е. когда $q \neq u$ на шаге 2A или когда $q \neq r$ на шаге 3 процедуры Partition. Наилучший случай с точки зрения минимизации обменов является также одним из наихудших в смысле асимптотического времени работы: когда массив изначально отсортирован. В этом случае обмены не выполняются. Наибольшее количество обменов осуществляется, когда n четно и входной массив имеет вид $n, n-2, n-4, \ldots, 4, 2, 1, 3, 5, \ldots, n-3, n-1$. В этом случае выполняется $n^2/4$ обменов, и асимптотическое время работы алгоритма соответствует наихудшему случаю $\Theta(n^2)$.

Резюме

В этой и предыдущей главах вы познакомились с четырьмя алгоритмами поиска и четырьмя алгоритмами сортировки. Давайте подытожим их свойства в двух таблицах. Поскольку три алгоритма поиска в главе 2, "Описание и оценка компьютерных алгоритмов", были всего лишь вариациями, в качестве представительного алгоритма для линейного поиска достаточно рассмотреть Better-Linear-Search или Sentinel-Linear-Search.

Алгоритмы поиска

Алгоритм	Время работы в наихудшем случае	Время работы в наилучшем случае	Требует ли отсортированного входного массива
Линейный поиск	$\Theta(n)$	$\Theta(1)$	Нет
Бинарный поиск	$\Theta(\lg n)$	$\Theta(1)$	Да

Алгоритмы сортировки

Алгоритм сортировки	Время работы в наихудшем случае	Время работы в наилучшем случае	Обменов в наихудшем случае	Выполняется ли сортировка на месте
Выбором	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	Да
Вставкой	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	Да
Слиянием	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$	Нет
Быстрая	$\Theta(n^2)$	$\Theta(n \lg n)$	$\Theta(n^2)$	Да

В таблицах не показано время работы в среднем случае, поскольку, за исключением быстрой сортировки, оно соответствует времени работы в наихудшем случае. Как мы видели, в среднем случае (в предположении случайно упорядоченного массива) быстрая сортировка работает за время, составляющее всего лишь $\Theta(n \lg n)$.

Как соотносятся эти алгоритмы сортировки на практике? Я реализовал их на языке программирования C++ и применил к массивам 4-байтовых целых чисел на двух разных компьютерах: на моем MacBook Pro (на котором я писал эту книгу) с процессором Intel Core 2 Duo 2.4 ГГц и 4 ГБайт ОЗУ под управлением Mac OS 10.6.8 и на Dell PC (сервер моего веб-сайта) с процессором 3.2 ГГц Intel Pentium 4 и 1 ГБайт ОЗУ под управлением Linux версии 2.6.22.14. Код компилирован компилятором g++ и уровнем оптимиза ции -O3. Каждый алгоритм тестировался на массиве размером до 50000 элементов; каждый массив изначально был отсортирован в обратном порядке. Я усреднял времена работы по 20 запускам для каждого алгоритма и каждого размера массива.

Используя массивы, отсортированные в обратном порядке, я сознательно получал наихудший случай для сортировки вставкой и быстрой сортировки. Быструю сортировку я реализовал в двух вариантах: ***детерминистическую*** (т.е. действия которой всегда выполняются одинаково), которая всегда выбирает в качестве опорного последний элемент подмассива $A[p..r]$, и рандомизированную, которая перед разбиением меняет $A[r]$ со случайно выбранным элементом из $A[p..r]$ (метод медианы трех случайных элементов я реализовывать не стал).

Рандомизированная быстрая сортировка оказалась чемпионкой для $n \geq 64$ на обоих компьютерах. Вот отношение времени работы других алгоритмов ко времени работы рандомизированной быстрой сортировки для входных данных разных размеров.

MacBook Pro

	n						
Алгоритм сортировки	50	100	500	1000	5000	10000	50000
Выбором	1.34	2.13	8.04	13.31	59.07	114.24	537.42
Вставкой	1.08	2.02	6.15	11.35	51.86	100.38	474.29
Слиянием	7.58	7.64	6.93	6.87	6.35	6.20	6.27
Детерминистическая быстрая	1.02	1.63	6.09	11.51	52.02	100.57	475.34

Dell PC

	n						
Алгоритм сортировки	50	100	500	1000	5000	10000	50000
Выбором	0.76	1.60	5.46	12.23	52.03	100.79	496.94
Вставкой	1.01	1.66	7.68	13.90	68.34	136.20	626.44
Слиянием	3.21	3.38	3.57	3.33	3.36	3.37	3.15
Детерминистическая быстрая	1.12	1.37	6.52	9.30	47.60	97.45	466.83

Рандомизированная быстрая сортировка выглядит очень хорошо, но и ее можно превзойти. Вспомним, что сортировка вставкой очень хороша, когда элемент не должен

двигаться по массиву далеко. Но как только размер подзадачи в рекурсивном алгоритме опускается до некоторого значения k, то никакой элемент не будет перемещаться более чем на $k-1$ позиций. Поэтому вместо того, чтобы продолжать рекурсивный вызов рандомизированной быстрой сортировки для подзадач небольших размеров, можно воспользоваться сортировкой вставкой. Действительно, такой гибридный метод позволяет превзойти рандомизированную быструю сортировку. Я обнаружил, что на моем MacBook Pro оптимальным для переключения алгоритмов был размер подзадачи, равный 22, а на Dell PC оптимальным был размер 17 элементов. Ниже приведены отношения времен работы гибридного и рандомизированного алгоритмов быстрой сортировки на обеих машинах для задач разных размеров.

				n			
Машина	50	100	500	1000	5000	10000	50000
MacBook Pro	0.55	0.56	0.60	0.60	0.62	0.63	0.66
Dell PC	0.53	0.58	0.60	0.58	0.60	0.64	0.64

Можно ли превзойти время сортировки $\Theta(n \lg n)$? Это зависит от того, что и как сортируется. Мы увидим в главе 4, “Нижняя граница времени сортировки и как ее превзойти”, что если единственным способом определения порядка размещения элементов является их сравнение, то превзойти время $\Theta(n \lg n)$ невозможно. Но если имеется дополнительная информация о сортируемых элементах, то могут быть ситуации, когда это время сортировки можно превзойти.

Дальнейшее чтение

В CLRS [4] рассматриваются сортировка вставкой, сортировка слиянием, а также детерминистическая и рандомизированная быстрая сортировки. Но суперкнигой о сортировке и поиске остается том 3 *Искусства программирования* Д. Кнута (Donald Knuth) [12]. Здесь применим совет из главы 1 нашей книги: если вы не боитесь трудностей и математических сложностей, обратитесь к *Искусству программирования*.

4... Нижняя граница времени сортировки и как ее превзойти

В предыдущей главе вы познакомились с четырьмя алгоритмами для сортировки n элементов в массиве. Два из них, сортировка выбором и сортировка вставкой, имеют время работы в наихудшем случае, равное $\Theta(n^2)$, что не очень-то хорошо. Один из алгоритмов, алгоритм быстрой сортировки, в наихудшем случае имеет время работы $\Theta(n^2)$, но в среднем случае выполняет сортировку только за время $\Theta(n \lg n)$. Сортировка слиянием имеет время работы $\Theta(n \lg n)$ во всех случаях. На практике наиболее эффективной является быстрая сортировка, но если вы хотите абсолютно гарантированно защититься от наихудшего случая, следует использовать сортировку слиянием.

Но насколько хорошим является время $\Theta(n \lg n)$? Нельзя ли разработать алгоритм сортировки, который в наихудшем случае превзойдет время $\Theta(n \lg n)$? Ответ зависит от правил игры: как алгоритм сортировки может использовать ключи сортировки для определения порядка сортировки?

В этой главе мы увидим, что при определенном наборе правил превзойти время работы $\Theta(n \lg n)$ невозможно. Затем мы рассмотрим два алгоритма сортировки, сортировку подсчетом и карманную сортировку, которые используют другие правила, а потому в состоянии выполнять сортировку за время всего лишь $\Theta(n)$.

Правила сортировки

Если рассмотреть, как четыре алгоритма из предыдущей главы используют ключи сортировки, то можно увидеть, что они определяют порядок сортировки, основываясь только на сравнении пары ключей. Все принимаемые ими решения имеют вид "если ключ сортировки этого элемента меньше, чем ключ сортировки другого элемента, то то-то, а в противном случае либо сделать что-то еще, либо ничего не делать". Вы можете подумать, что алгоритм сортировки может принимать решения *только* такого вида. А какие еще виды решений он в состоянии принимать?

Чтобы убедиться в том, что возможны и другие виды решений, давайте рассмотрим очень простой пример. Предположим, что мы знаем две вещи о сортируемых элементах, а именно — что каждый ключ сортировки является либо единицей, либо двойкой и что элементы состоят только из ключей сортировки, не имея никаких сопутствующих данных. В этой простой ситуации можно сортировать n элементов за время $\Theta(n)$, превзойдя алгоритмы со временем работы $\Theta(n \lg n)$ из предыдущей главы. Каким образом? Начнем с того, что пройдем по всем элементам и подсчитаем, сколько среди них единиц, — скажем, это k элементов. Тогда можно вновь пройти через массив, устанавливая значение 1 в первых k позициях, а затем устанавливая значение 2 в остальных $n-k$ позициях. Вот как выглядит соответствующая процедура.

Процедура REALLY-SIMPLE-SORT(A,*n*)

Вход:

- *A*: массив, все элементы которого имеют значения 1 или 2.
- *n*: количество сортируемых элементов *A*.

Результат: элементы *A* отсортированы в неубывающем порядке.

1. Установить *k* равным нулю.
2. Для $i = 1$ до *n*:

   A. Если $A[i] = 1$, увеличить *k* на единицу.
3. Для $i = 1$ до *k*:

   A. Установить $A[i]$ равным 1.
4. Для $i = k + 1$ до *n*:

   A. Установить $A[i]$ равным 2.

Шаги 1 и 2 подсчитывают количество единиц, увеличивая счетчик *k* для каждого элемента $A[i]$, равного 1. Шаг 3 заполняет подмассив $A[1..k]$ единицами, а шаг 4 заполняет остальные позиции, $A[k+1..n]$, двойками. Легко видеть, что эта процедура выполняется за время $\Theta(n)$: первый цикл выполняет *n* итераций, как и два последних цикла вместе; каждая итерация каждого цикла выполняется за постоянное время.

Обратите внимание, что процедура REALLY-SIMPLE-SORT никогда не сравнивает два элемента массива один с другим. Она сравнивает каждый элемент массива со значением 1, но не с другим элементом массива. Так что, как видите, в такой ограниченной ситуации возможна сортировка без сравнения пар ключей сортировки.

Нижняя граница сортировки сравнением

Теперь, когда у вас есть некоторое представление о том, как могут варьироваться правила игры, давайте рассмотрим, насколько быстрой может быть сортировка.

Определим ***сортировку сравнением*** как любой алгоритм сортировки, который определяет порядок сортировки только путем сравнения пар элементов. Сортировкой сравнением являются четыре алгоритма сортировки из предыдущей главы, но не алгоритм REALLY-SIMPLE-SORT.

Вот нижняя граница сортировки сравнением.

В наихудшем случае любой алгоритм сортировки сравнением требует для сортировки *n* элементов $\Omega(n \lg n)$ сравнений пар элементов.

Вспомним, что Ω-обозначение дает нижнюю границу, так что мы, по сути, говорим “для достаточно больших *n* любой алгоритм сортировки сравнением требует в наихудшем случае выполнения по крайней мере $cn \lg n$ сравнений для некоторой константы *c*”. Поскольку каждое сравнение выполняется по меньшей мере за постоянное время, это дает

нам нижнюю границу $\Omega(n \lg n)$ времени сортировки n элементов при условии, что используется алгоритм сортировки сравнением.

Важно иметь ясное представление о нижней границе. Первое — она говорит что-то только о наихудшем случае. Вы всегда можете сделать алгоритм сортировки работающим за линейное время в наилучшем случае: просто заявить, что наилучший случай — это когда массив уже отсортирован, и просто проверить, что каждый элемент (за исключением последнего) не превышает его преемника в массиве. Это легко сделать за время $\Theta(n)$. Если вы обнаружите, что каждый элемент не превышает его преемника, то сортировка выполнена. Однако в наихудшем случае $\Omega(n \lg n)$ сравнений являются необходимыми. Мы называем эту нижнюю границу ***экзистенциальной*** нижней границей, потому что она утверждает, что существуют входные данные, которые требуют $\Omega(n \lg n)$ сравнений. Другой тип нижней границы — ***универсальная*** нижняя граница, которая применима ко всем входным данным. В случае сортировки единственной универсальной нижней границей является $\Omega(n)$, поскольку мы должны взглянуть на каждый элемент по крайней мере один раз. Обратите внимание, что в предыдущем предложении я не сказал, к *чему* относится $\Omega(n)$. Имел ли я в виду $\Omega(n)$ сравнений или время работы $\Omega(n)$? Я подразумевал время, поскольку мы должны проверить каждый элемент, даже если не сравниваем элементы попарно.

Вторая важная вещь, которую следует сказать о нижней границе, действительно замечательна: это то, что нижняя граница не зависит от конкретного алгоритма, лишь бы этот алгоритм являлся алгоритмом сортировки сравнением. Нижняя граница применяется к *любому* алгоритму сортировки сравнением, независимо от того, насколько простым или сложным он является. Нижняя граница применима ко всем алгоритмам сортировки сравнением, которые уже были изобретены или еще только будут открыты в будущем. Она справедлива даже для тех алгоритмов сортировки сравнением, которые никогда не будут обнаружены человечеством!

Сортировка подсчетом

Мы уже видели, как превзойти нижнюю границу при очень ограниченных условиях, когда имеется только два возможных значения ключа сортировки, а каждый элемент состоит только из ключа сортировки, без сопутствующих данных. В этом ограниченном случае n элементов можно отсортировать за время $\Theta(n)$, без сравнения пар элементов.

Метод Really-Simple-Sort можно обобщить на случай m различных возможных значений ключей сортировки, которые являются целыми числами из диапазона, представляющего собой m последовательных целых чисел (скажем, от 0 до $m-1$), а элементы при этом могут иметь сопутствующие данные.

Вот в чем заключается идея. Предположим, мы знаем, что ключами сортировки являются целые числа в диапазоне от 0 до $m-1$. Кроме того, предположим, мы знаем, что ровно у трех элементов ключ сортировки равен 5 и что у шести элементов ключи сортировки меньше 5 (т.е. находятся в диапазоне от 0 до 4). Тогда мы знаем, что в отсортированном массиве элементы с ключом сортировки, равным 5, должны занимать позиции 7, 8 и 9.

Обобщая, если мы знаем, что у k элементов ключи сортировки равны x, а у l элементов ключи сортировки меньше x, то элементы с ключами сортировки, равными x, в отсортированном массиве должны занимать позиции от $l+1$ до $l+k$. Таким образом, нам надо для каждого возможного значения ключа сортировки вычислить, у какого количества элементов ключи сортировки ключей меньше этого значения и сколько имеется элементов с данным значением ключа сортировки.

Мы можем вычислить, у скольких элементов ключи сортировки меньше каждого из возможных значений ключа, если начнем с того, что вычислим, у какого количества элементов ключи сортировки равны заданному значению. Начнем нашу работу с решения этой задачи.

Процедура COUNT-KEYS-EQUAL(A,n,m)

Вход:

- A: массив целых чисел в диапазоне от 0 до $m-1$.
- n: количество элементов в массиве A.
- m: определяет диапазон значений в массиве A.

Выход: массив $equal[0..m-1]$, такой, что $equal[j]$ содержит количество элементов массива A, равных j, для $j=0,1,2,\dots,m-1$.

1. Пусть $equal[0..m-1]$ представляет собой новый массив.
2. Установить все значения массива *equal* равными нулю.
3. Для $i=1$ до n:

   A. Установить значение переменной *key* равным $A[i]$.

   B. Увеличить $equal[key]$ на единицу.
4. Вернуть массив *equal*.

Обратите внимание, что процедура COUNT-KEYS-EQUAL никогда не сравнивает ключи сортировки один с другим. Она использует ключи сортировки только в качестве индекса в массиве *equal*. Поскольку первый цикл (неявный цикл на шаге 2) делает m итераций, второй цикл (на шаге 3) делает n итераций, и каждая итерация каждого цикла выполняется за константное время, процедура COUNT-KEYS-EQUAL выполняется за время $\Theta(n+m)$. Если m является константой, то время работы COUNT-KEYS-EQUAL равно $\Theta(n)$.

Теперь мы можем использовать массив *equal* для выяснения, у какого количества элементов ключи сортировки меньше каждого возможного значения.

Процедура COUNT-KEYS-LESS(*equal*,m)

Вход:

- *equal*: массив, возвращаемый вызовом процедуры COUNT-KEYS-EQUAL.
- m: определяет диапазон индексов массива *equal* — от 0 до $m-1$.

Выход: массив $less[0..m-1]$, такой, что для $j = 0,1,2,\dots,m-1$ элемент $less[j]$ содержит сумму $equal[0]+equal[1]+\dots+equal[j-1]$.

1. Пусть $less[0..m-1]$ представляет собой новый массив.
2. Установить $less[0]$ равным нулю.
3. Для $j = 1$ до $m-1$:

   A. Установить $less[j]$ равным $less[j-1]+equal[j-1]$.
4. Вернуть массив *less*.

В предположении, что $equal[j]$ дает точное количество элементов, ключи сортировки которых равны j, для $j = 0,1,\dots,m-1$, можно использовать следующий инвариант цикла, чтобы показать, что по завершении работы процедуры Count-Keys-Less значение $less[j]$ равно количеству ключей сортировки, меньших j.

В начале каждой итерации цикла на шаге 3 значение $less[j-1]$ равно количеству ключей сортировки, меньших $j-1$.

Расписать части инициализации, сохранения и завершения я предоставляю читателю. Можно легко увидеть, что процедура Count-Keys-Less выполняется за время $\Theta(m)$. И она, определенно, не выполняет ни одного сравнения ключей одного с другим.

Рассмотрим пример. Предположим, что $m = 7$, так что все ключи сортировки являются целыми числами в диапазоне от 0 до 6, и у нас есть следующий массив A с $n = 10$ элементами: $A = \langle 4,1,5,0,1,6,5,1,5,3 \rangle$. Тогда $equal = \langle 1,3,0,1,1,3,1 \rangle$ и $less = \langle 0,1,4,4,5,6,9 \rangle$. Поскольку $less[5] = 6$, а $equal[5] = 3$ (вспомните, что индексы массивов *less* и *equal* начинаются с 0, а не с 1), по окончании сортировки позиции с 1 по 6 должны содержать значения ключей, меньшие, чем 5, а в позициях 7, 8 и 9 должно содержаться значение ключа, равное 5.

Когда у нас есть массив *less*, мы можем создать отсортированный массив, хотя и не на месте.

Процедура Rearrange(A,less,n,m)

Вход:

- A: массив целых чисел в диапазоне от 0 до $m-1$.
- *less*: массив, возвращаемый процедурой Count-Keys-Less.
- n: количество элементов в массиве A.
- m: определяет диапазон значений элементов в массиве A.

Выход: массив B, содержащий элементы массива A в отсортированном порядке.

1. Пусть $B[1..n]$ и $next[0..m-1]$ — новые массивы.
2. Для $j = 0$ до $m-1$:

   A. Установить $next[j]$ равным $less[j]+1$.

	0	1	2	3	4	5	6
less	0	1	4	4	5	6	9
next	1	2	5	5	6	7	10

	1	2	3	4	5	6	7	8	9	10
A	4	1	5	0	1	6	5	1	5	3
B										

↓

	0	1	2	3	4	5	6
next	1	2	5	5	7	7	10

	1	2	3	4	5	6	7	8	9	10
A	4	1	5	0	1	6	5	1	5	3
B						4				

↓

	0	1	2	3	4	5	6
next	1	3	5	5	7	7	10

	1	2	3	4	5	6	7	8	9	10
A	4	1	5	0	1	6	5	1	5	3
B		1				4				

↓

	0	1	2	3	4	5	6
next	1	3	5	5	7	8	10

	1	2	3	4	5	6	7	8	9	10
A	4	1	5	0	1	6	5	1	5	3
B		1				4	5			

↓

	0	1	2	3	4	5	6
next	2	3	5	5	7	8	10

	1	2	3	4	5	6	7	8	9	10
A	4	1	5	0	1	6	5	1	5	3
B	0	1				4	5			

↓

	0	1	2	3	4	5	6
next	2	4	5	5	7	8	10

	1	2	3	4	5	6	7	8	9	10
A	4	1	5	0	1	6	5	1	5	3
B	0	1	1			4	5			

↓

	0	1	2	3	4	5	6
next	2	4	5	5	7	8	11

	1	2	3	4	5	6	7	8	9	10
A	4	1	5	0	1	6	5	1	5	3
B	0	1	1			4	5			6

↓

	0	1	2	3	4	5	6
next	2	4	5	5	7	9	11

	1	2	3	4	5	6	7	8	9	10
A	4	1	5	0	1	6	5	1	5	3
B	0	1	1			4	5	5		6

↓

	0	1	2	3	4	5	6
next	2	5	5	5	7	9	11

	1	2	3	4	5	6	7	8	9	10
A	4	1	5	0	1	6	5	1	5	3
B	0	1	1	1		4	5	5		6

↓

	0	1	2	3	4	5	6
next	2	5	5	5	7	10	11

	1	2	3	4	5	6	7	8	9	10
A	4	1	5	0	1	6	5	1	5	3
B	0	1	1	1		4	5	5	5	6

↓

	0	1	2	3	4	5	6
next	2	5	5	6	7	10	11

	1	2	3	4	5	6	7	8	9	10
A	4	1	5	0	1	6	5	1	5	3
B	0	1	1	1	3	4	5	5	5	6

3. Для $i = 1$ до n:
   A. Установить значение *key* равным $A[i]$.
   B. Установить значение *index* равным $next[key]$.
   C. Установить $B[index]$ равным $A[i]$.
   D. Увеличить значение $next[key]$ на единицу.
4. Вернуть массив B.

Приведенный далее рисунок показывает, как процедура Rearrange перемещает элементы из массива A в массив B так, чтобы они в конечном итоге оказались в массиве B в отсортированном порядке. В верхней части рисунка показаны массивы *less*, *next*, A и B перед первой итерацией цикла на шаге 3, а в последующих частях показаны массивы *next*, A и B после каждой очередной итерации. Серым цветом показаны элементы, скопированные в массив B.

Идея заключается в том, что при проходе по массиву A от начала до конца $next[j]$ указывает индекс элемента в массиве B, в который должен быть помещен очередной элемент массива A с ключом j. Вспомните, что если l элементов имеют ключи сортировки, меньшие, чем x, то k элементов с ключом сортировки, равным x, должны занимать позиции от $l+1$ до $l+k$. Цикл на шаге 2 выполняет установку значений массива *next* так, что изначально $next[j] = l+1$, где $l = less[j]$. Цикл на шаге 3 обходит массив A от начала до конца. Для каждого элемента $A[i]$ шаг 3A сохраняет значение $A[i]$ в переменной *key*, шаг 3B вычисляет значение *index*, которое представляет собой индекс в массиве B, где должно быть сохранено значение $A[i]$, а шаг 3C переносит $A[i]$ в эту позицию в массиве B. Поскольку следующий элемент в массиве A, который имеет тот же ключ сортировки, что и $A[i]$ (если таковой имеется), должен быть сохранен в следующей позиции массива B, шаг 3D увеличивает значение $next[key]$ на единицу.

Каково время работы процедуры Rearrange? Цикл на шаге 2 выполняется за время $\Theta(m)$, а цикл на шаге 3 — за время $\Theta(n)$. Следовательно, процедура Rearrange, как и процедура Count-Keys-Equal, имеет время работы $\Theta(n+m)$, что равно $\Theta(n)$, если m является константой.

Теперь мы можем собрать все три процедуры вместе для создания процедуры ***сортировки подсчетом***.

Процедура Counting-Sort(A,n,m)

Вход:

- A: массив целых чисел в диапазоне от 0 до $m-1$.
- n: количество элементов в массиве A.
- m: определяет диапазон значений в массиве A.

Выход: массив B, содержащий элементы массива A в отсортированном порядке.

1. Вызвать процедуру Count-Keys-Equal(A,n,m) и сохранить ее результат как массив *equal*.
2. Вызвать процедуру Count-Keys-Less(*equal*,m) и сохранить ее результат как массив *less*.
3. Вызвать процедуру Rearrange(A,*less*,n,m) и сохранить ее результат как массив B.
4. Вернуть массив B.

Исходя из времени работы процедур Count-Keys-Equal ($\Theta(n+m)$), Count-Keys-Less ($\Theta(m)$) и Rearrange ($\Theta(n+m)$), можно сделать вывод, что процедура Counting-Sort выполняется за время $\Theta(n+m)$, или просто $\Theta(n)$, если m представляет собой константу. Сортировка подсчетом превосходит нижнюю границу $\Omega(n \lg n)$ сортировки сравнением, потому что она никогда не сравнивает ключи сортировки один с другим. Вместо этого она использует ключи сортировки для индексирования массивов, что вполне реально, когда ключи сортировки являются небольшими целыми значениями. Если ключи сортировки представляют собой действительные числа с дробной частью или, например, строки символов, то в таком случае использовать сортировку подсчетом нельзя.

Вы можете заметить, что процедура предполагает наличие в элементах только ключей сортировки без каких-либо сопутствующих данных. Да, я обещал, что в отличие от Really-Simple-Sort процедура Counting-Sort допускает наличие сопутствующих данных. Этого легко добиться, достаточно только модифицировать шаг 3C процедуры Rearrange так, чтобы он копировал весь элемент, а не только ключ сортировки. Вы могли также обратить внимание на то, что мои процедуры несколько неэффективно используют массивы. Да, массивы *equal*, *less* и *next* можно объединить в один массив, но эту задачу я оставляю в качестве задания читателю.

Я постоянно отмечал, что время работы равно $\Theta(n)$, если m является константой. Но когда m является константой? Один из примеров — сортировка студентов по уровню успеваемости. Например, уровень успеваемости может принимать значения от 0 до 10, но количество студентов при этом варьируется. Я вполне мог бы использовать сортировку подсчетом для n студентов за время $\Theta(n)$, так как значение $m = 11$ (вспомните, что сортируемый диапазон значений — от 0 до $m-1$) является константой.

На практике однако сортировка подсчетом оказывается полезной в качестве части другого алгоритма сортировки — поразрядной сортировки. В дополнение к линейному времени работы при константном значении m сортировка подсчетом имеет еще одно важное свойство: она является ***устойчивой***. В случае устойчивой сортировки элементы с одним и тем же ключом сортировки оказываются в выходном массиве в том же порядке, что и во входном. Другими словами, устойчивая сортировка, встречая два элемента с равными ключами, разрешает неоднозначность, помещая в выходной массив первым тот элемент, который появляется первым во входном массиве. Понять, почему сортировка подсчетом является устойчивой, можно глядя на цикл на шаге 3 процедуры Rearrange. Если два элемента A имеют один и тот же ключ сортировки, скажем, *key*, то процедура увеличивает *next*[*key*] сразу же после переноса в массив B элемента, который ранее был в A. Таким

образом, к моменту перемещения элемента, который появляется в A позже, этот элемент будет помещаться в массив B в позицию с бо́льшим индексом.

Поразрядная сортировка

Предположим, что вам нужно отсортировать строки символов некоторой фиксированной длины. Например сейчас я пишу этот раздел, сидя в самолете, и когда я делал заказ билета, мой код подтверждения был XI7FS6. Все коды подтверждения авиакомпании имеют вид строк из шести символов, каждый из которых является буквой или цифрой. Каждый символ может принимать 36 значений (26 букв плюс 10 цифр), так что всего имеется $36^6 = 2\,176\,782\,336$ возможных кодов подтверждения. Хотя это и константа, но она слишком велика, чтобы авиакомпания использовала для сортировки кодов сортировку подсчетом.Чтобы получить для каждого кода конкретное число, можно перевести каждый из 36 символов в числовой код, имеющий значение от 0 до 35. Код цифры является самой этой цифрой (так, код цифры 5 представляет собой число 5), а коды для букв начинаются с 10 для A и заканчиваются 35 для Z.

Теперь давайте немного упростим задачу и предположим, что каждый код подтверждения состоит только из двух символов (не беспокойтесь: мы вскоре вернемся к шести символам). Хотя можно воспользоваться сортировкой подсчетом с $m = 36^2 = 1296$, мы вместо этого используем ее *дважды* с $m = 36$. В первый раз в качестве ключа сортировки используем *правый* символ, а затем вновь отсортируем результат, но теперь в качестве ключа используем *левый* символ. Мы выбираем сортировку подсчетом, потому что она хорошо работает при относительно небольших m и потому что она устойчива.

Предположим, например, что у нас есть двухсимвольные коды <F6, E5, R6, X6, X2, T5, F2, T3>. После сортировки подсчетом по правому символу мы получаем коды в следующем порядке: <X2, F2, T3, E5, T5, F6, R6, X6>. Обратите внимание, что, поскольку сортировка подсчетом устойчива, а X2 в исходном порядке идет до F2, после сортировки по правому символу X2 продолжает находиться перед F2. Теперь отсортируем результат по левому символу, вновь используя сортировку подсчетом, и получим то, что хотели: <E5, F2, F6, R6, T3, T5, X2, X6>.

Что бы произошло, если бы мы сначала выполнили сортировку по левому символу? После сортировки подсчетом по левому символу мы бы получили <E5, F6, F2, R6, T5, T3, X6, X2>, а затем после сортировки подсчетом по правому символу был бы получен неверный окончательный результат <F2, X2, T3, E5, T5, F6, R6, X6>.

Почему работа справа налево приводит к правильному результату? Важное значение имеет использование устойчивой сортировки; это может быть сортировка подсчетом или любая иная, но устойчивая сортировка. Предположим, что мы работаем с символами в i-й позиции и что по всем $i-1$ позициям справа массив был отсортирован. Рассмотрим две любые сортировки ключей. Если они отличаются в i-й позиции, то их $i-1$ позиций справа значений не имеют: устойчивый алгоритм сортировки по i-й позиции разместит их в верном порядке. Если же, с другой стороны, они имеют один и тот же символ в i-й позиции,

то первым должен быть код, символ которого идет первым в $i-1$ позиции. Но применение метода устойчивой сортировки гарантирует получение именно этого результата.

Вернемся к нашим шестизначным кодам подтверждения и посмотрим, как будут отсортированы коды, которые изначально находились в порядке <XI7FS6, PL4ZQ2, JI8FR9, XL8FQ6, PY2ZR5, KV7WS9, JL2ZV3, KI4WR2>. Пронумеруем символы справа налево от 1 до 6. Тогда последовательность кодов после выполнения устойчивой сортировки по i-му символу имеет следующий вид.

i	Последовательность после сортировки
1	<PL4ZQ2, KI4WR2, JL2ZV3, PY2ZR5, XI7FS6, XL8FQ6, JI8FR9, KV7WS9>
2	<PL4ZQ2, XL8FQ6, KI4WR2, PY2ZR5, JI8FR9, XI7FS6, KV7WS9, JL2ZV3>
3	<XL8FQ6, JI8FR9, XI7FS6, KI4WR2, KV7WS9, PL4ZQ2, PY2ZR5, JL2ZV3>
4	<PY2ZR5, JL2ZV3, KI4WR2, PL4ZQ2, XI7FS6, KV7WS9, XL8FQ6, JI8FR9>
5	<KI4WR2, XI7FS6, JI8FR9, JL2ZV3, PL4ZQ2, XL8FQ6, KV7WS9, PY2ZR5>
6	<JI8FR9, JL2ZV3, KI4WR2, KV7WS9, PL4ZQ2, PY2ZR5, XI7FS6, XL8FQ6>

Обобщая, в алгоритме ***поразрядной сортировки*** мы предполагаем, что каждый ключ сортировки можно рассматривать как d-значное число, каждая цифра которого находится в диапазоне от 0 до $m-1$. Мы поочередно используем устойчивую сортировку для каждой цифры справа налево. Если в качестве устойчивой применяется сортировка подсчетом, то время сортировки по одной цифре составляет $\Theta(m+n)$, а время сортировки по всем d цифрам — $\Theta(d(m+n))$. Если m является константой (как в примере с кодами подтверждения $m = 36$), то время работы поразрядной сортировки становится равным $\Theta(dn)$. Если d также представляет собой константу (например, 6 в случае кодов подтверждения), то время работы поразрядной сортировки превращается в просто $\Theta(n)$.

Когда поразрядная сортировка использует сортировку подсчетом для упорядочения по каждой цифре, она никогда не сравнивает два ключа сортировки один с другим. Она использует отдельные цифры для индексирования массивов в сортировке подсчетом. Вот почему поразрядная сортировка, как и сортировка подсчетом, преодолевает нижнюю границу $\Omega(n \lg n)$ сортировки сравнением.

Дальнейшее чтение

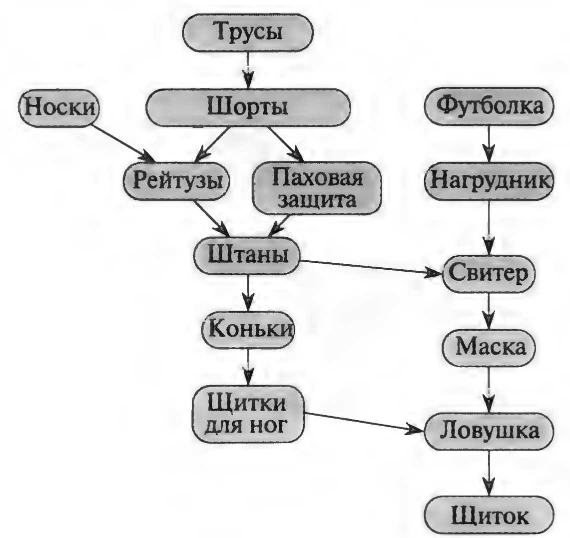
В главе 8 в CLRS [4] весь материал данной главы охвачен более подробно и широко.

5... Ориентированные ациклические графы

Когда-то я был неплохим хоккеистом. Несколько лет я был вратарем, но потом мой интерес к игре угас, и я забросил клюшку и щиток. Но однажды, после более чем семилетнего перерыва, случилось так, что мне удалось поучаствовать в паре игр.

Самой большой моей проблемой оказалась даже не игра — хотя я и знал, что на поле я буду выглядеть ужасно, — а то, что я напрочь позабыл, как одевается вся эта вратарская сбруя (а ее немало — до 15–20 кг). Готовясь к выходу на поле, все это надо нацепить на себя в правильном порядке. Например, поскольку я правша, на левую руку я одеваю огромную рукавицу для ловли шайбы; она называется ловушкой (catch glove). После того как я одену эту перчатку, я больше не смогу ничего одеть на верхнюю часть тела — настолько она велика.

Словом, когда я готовился к игре, мне пришлось нарисовать для себя диаграмму, показывающую, что и в каком порядке мне следует одевать (эта диаграмма приведена ниже). Стрелка от A к B указывает на то, что A надо одеть до B. Например, я должен надеть нагрудник до свитера. Очевидно, что ограничение “следует одеть до” является ***транзитивным***: если A следует одеть до B, а B надо одеть до C, то A должен быть одет до C. Поэтому нагрудник надо надеть до свитера, маски, ловушки и щитка.

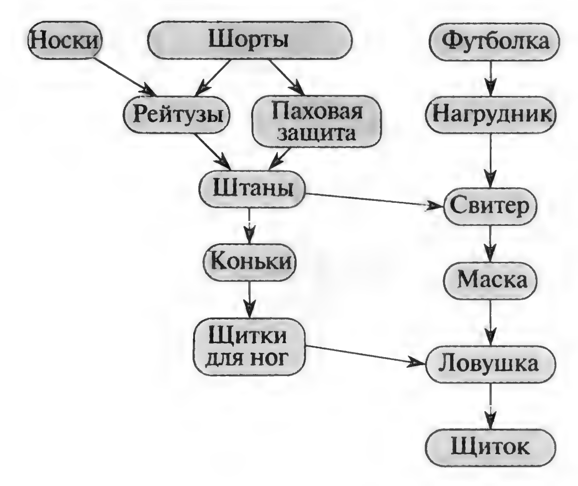


Некоторые пары вещей можно одевать в любом порядке. Например, можно одеть носки как до, так и после нагрудника.

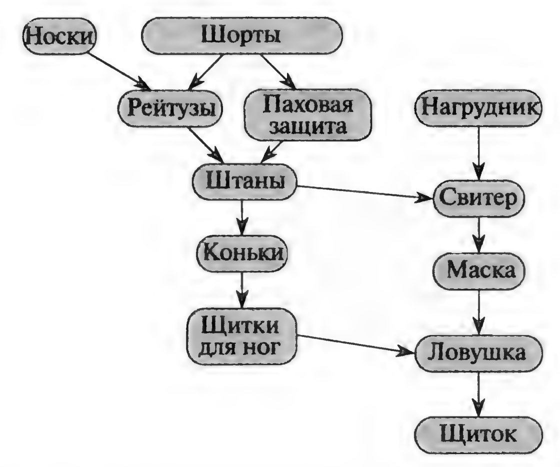
Мне надо было определить порядок одевания. Нарисовав диаграмму, я составил список из всех элементов экипировки, расположив их так, чтобы не нарушалось ни одно из ограничений “следует одеть до”. Я обнаружил, что таких списков может быть несколько; под диаграммой представлены три из них.

Порядок 1	Порядок 2	Порядок 3
Трусы	Трусы	Носки
Шорты	Футболка	Футболка
Паховая защита	Шорты	Трусы
Носки	Паховая защита	Нагрудник
Рейтузы	Нагрудник	Шорты
Штаны	Носки	Рейтузы
Коньки	Рейтузы	Паховая защита
Щитки для ног	Штаны	Штаны
Футболка	Свитер	Коньки
Нагрудник	Маска	Щитки для ног
Свитер	Коньки	Свитер
Маска	Щитки для ног	Маска
Ловушка	Ловушка	Ловушка
Щиток	Щиток	Щиток

Как я получил эти списки? Вот как я получил второй из них. Я искал элемент, в который нет входящих стрелок, потому что такой элемент не обязан следовать после другого. Я выбрал в качестве первого элемента трусы, а затем, одев их (концептуально), я удалил их из диаграммы, что дало мне следующую диаграмму.



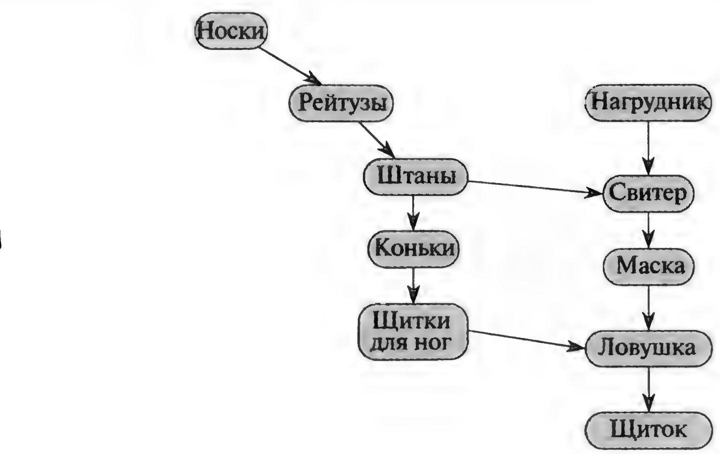
Затем я вновь выбрал элемент без входящей стрелки, в этот раз — футболку. Добавив ее к концу списка, я удалил ее из диаграммы, получив новую диаграмму.



И вновь я выбираю элемент без входящих стрелок (шорты) и выполняю те же действия, что и ранее.



После выбора паховой защиты диаграмма принимает следующий вид.



Таким образом я действовал до тех пор, пока не осталось ни одного элемента. Три списка, показанные на с. 80, представляют собой результат различных вариантов выбора элемента без входящих стрелок на исходной диаграмме.

Ориентированные ациклические графы

Приведенные диаграммы являются примерами ***ориентированных графов***, которые составлены из ***вершин***, соответствующих элементам экипировки вратаря, и ***ориентированных ребер***, показанных с помощью стрелок. Каждое ориентированное ребро представляет собой упорядоченную пару вида (u,v), где u и v — вершины. Например, крайним слева ребром в ориентированном графе на с. 80 является ребро (носки, рейтузы). Если ориентированный граф содержит ориентированное ребро (u,v), мы говорим, что вершина v является ***смежной*** с вершиной u и что ребро (u,v) ***покидает*** u и ***входит*** в v, так что вершина, помеченная как "рейтузы", является смежной с вершиной "носки", а ребро (носки, рейтузы) покидает вершину "носки" и входит в вершину "рейтузы".

Ориентированные графы, которые мы видели в этой главе, обладают еще одним свойством: из вершины такого графа нельзя попасть в нее же, пройдя по некоторой ненулевой последовательности ребер. Такой ориентированный граф называется ***ориентированным ациклическим графом***. Он называется ациклическим, поскольку в нем отсутствуют "циклы", т.е. пути из вершины обратно в нее же (более точное определение цикла мы дадим в этой главе позже).

Ориентированные ациклические графы идеально подходят для моделирования зависимостей, когда одна задача должна быть выполнена до другой. Другое использование для ориентированных ациклических графов — при планировании проектов, таких как строительство дома, например стены должны быть на месте до крыши. Или в кулинарии, где определенные шаги при приготовлении блюда должны происходить в установленном порядке, а для некоторых шагов их взаимный порядок совершенно не важен (мы рассмотрим пример такого ориентированного ациклического графа далее в этой главе).

Топологическая сортировка

Когда мне нужно было определить линейный порядок, в котором следует одевать вратарскую экипировку, мне нужно было выполнить "топологическую сортировку". Говоря точнее, ***топологическая сортировка*** ориентированного ациклического графа выполняет линейное упорядочение вершин, такое, что если (u,v) представляет собой ребро ориентированного ациклического графа, то в этом линейном упорядочении u находится перед v. Топологическая сортировка отличается от сортировок, рассматривавшихся нами в главах 3, "Алгоритмы сортировки и поиска", и 4, "Нижняя граница времени сортировки и как ее превзойти".

Линейное упорядочение, производимое топологической сортировкой, не обязательно единственное. Но вы уже знаете об этом, поскольку каждый из трех списков на с. 80 может быть получен топологической сортировкой.

Еще с одним применением топологической сортировки я столкнулся в своей работе программиста много лет назад. Мы создавали системы автоматизированного проектирования, которые среди прочего могли поддерживать библиотеки частей. Одни части могут содержать другие части, но циклические зависимости при этом не допускались: никакая часть не может в конечном счете содержать себя же. Нам потребовалось записать часть конструкции на ленту (я же сказал, что работа выполнялась много лет тому назад) так, чтобы каждая часть предшествовали другим частям, которые ее содержат. Если каждая часть является вершиной, а ребро (u,v) указывает, что часть v содержит часть u, то нам необходимо записывать части в порядке, сгенерированном топологической сортировкой.

Какая вершина является хорошим кандидатом для первого места в линейном упорядочении? Любая без входящих в нее ребер. Число ребер, входящих в вершину, называется ***входящей степенью*** вершины, так что мы можем начинать с любой вершины с нулевой входящей степенью. К счастью, каждый ориентированный ациклический граф должен иметь по крайней мере одну вершину с нулевой входящей степенью и по крайней мере одну вершину с нулевой ***исходящей степенью*** (у которой нет покидающих ее ребер), так как в противном случае в графе будет иметься цикл.

Итак, предположим, что мы выбираем любую вершину с нулевой входящей степенью — назовем ее вершиной u — и поместим ее в начале линейного упорядочения. Поскольку мы выбрали вершину u первой, все другие вершины будут размещены в линейном упорядочении после u. В частности, любая вершина v, смежная с u, должна находиться в линейном упорядочении после u. Следовательно, можно безопасно удалить из ориентированного ациклического графа вершину u и все исходящие из нее ребра, так как нам известно, что мы позаботились обо всех зависимостях, определяемых этими ребрами. Когда мы удалим вершину и исходящие из нее ребра из ориентированного ациклического графа, то что у нас останется? Другой ориентированный ациклический граф! В конце концов, мы не можем создать цикл, удаляя вершины и ребра. А раз так, мы можем повторять описанный процесс над получившимся ориентированным ациклическим графом, находя в нем вершину с нулевой входящей степенью, помещая ее в линейное упорядочение после находящихся там вершин, удаляя эту вершину и исходящие из нее ребра из ориентированного ациклического графа, и т.д.

Приведенная ниже процедура топологической сортировки использует эту идею, но вместо реального удаления вершин и ребер из ориентированного ациклического графа она просто отслеживает входящую степень каждой вершины, уменьшая ее на единицу для каждого концептуально удаляемого входящего ребра. Так как массив индексируется целыми числами, предположим, что мы идентифицируем каждую вершину уникальным целым числом в диапазоне от 1 до n. Поскольку процедуре необходимо быстро находить вершины с нулевой входящей степенью, она поддерживает входящие степени вершин в массиве *in-degree*, индексируемом вершинами, и список *next* всех вершин с нулевой входящей степенью. На шагах 1–3 выполняется инициализация массива *in-degree*, далее шаг 4 инициализирует список *next*, а шаг 5 обновляет массив *in-degree* и список *next* при концептуальном удалении вершин и ребер. Процедура может выбирать любую вершину в списке *next* в качестве очередной вершины линейного упорядочения.

Процедура TOPOLOGICAL-SORT(G)

Вход: *G*: ориентированный ациклический граф с вершинами, пронумерованными от 1 до *n*.

Выход: линейное упорядочение вершин, такое, что вершина *u* находится в нем до вершины *v*, если (u,v) является ребром графа.

1. Пусть *in-degree*$[1..n]$ представляет собой новый массив; кроме того, создадим пустое линейное упорядочение вершин.
2. Установить все значения элементов массива *in-degree* равными 0.
3. Для каждой вершины *u*:

   A. Для каждой вершины *v*, смежной с вершиной *u*:

      i. Увеличить *in-degree*$[v]$ на единицу.
4. Создать список *next*, состоящий из всех вершин *u*, таких, что *in-degree*$[u]=0$.
5. Пока список *next* не пустой, выполнять следующее.

   A. Удалить вершину из списка *next*, называя ее вершиной *u*.

   B. Добавить вершину *u* в конец линейного упорядочения.

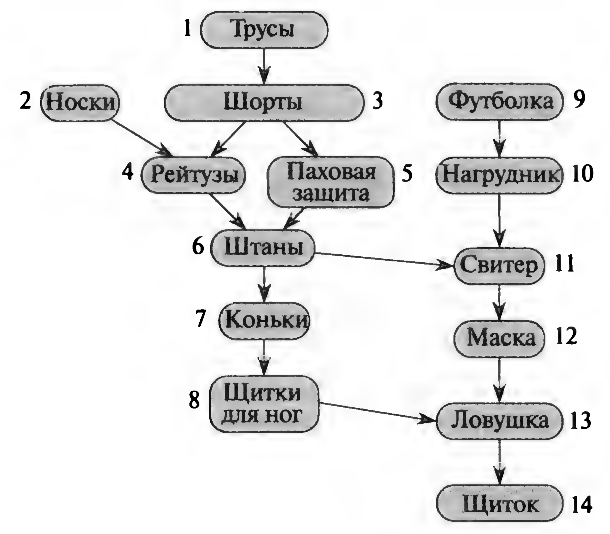
   C. Для каждой вершины *v*, смежной с вершиной *u*:

      i. Уменьшить *in-degree*$[v]$ на единицу.

      ii. Если *in-degree*$[v]=0$, внести вершину *v* в список *next*.
6. Вернуть линейное упорядочение.

Рассмотрим, как несколько первых итераций на шаге 5 работают с ориентированным ациклическим графом для одевания хоккейного вратаря. Чтобы выполнить процедуру TOPOLOGICAL-SORT над этим ориентированным ациклическим графом, нам нужно перенумеровать вершины, как показано на рисунке на с. 85. Нулевую входную степень имеют только вершины 1, 2 и 9 и при входе в цикл на шаге 5 список *next* содержит только эти три вершины. Чтобы получить первый из списков на с. 80, порядок вершин в списке *next* должен быть следующим: 1, 2, 9. Затем в первой итерации цикла на шаге 5 мы выбираем в качестве вершины *u* вершину 1 (трусы), удаляем ее из списка *next*, добавляем ее в конец изначально пустого линейного упорядочения, после чего уменьшаем на единицу элемент *in-degree*$[3]$ (шорты). Так как эта операция обнуляет значение *in-degree*$[3]$, мы вносим вершину 3 в список *next*. Будем считать, что при вставке вершины в список *next* мы вставляем его первой вершиной в списке. Такой список, в котором все вставки и удаления выполняются в одном конце, известен как ***стек*** и напоминает стопку тарелок, из которой всегда берется верхняя тарелка и в которую тарелка кладется на вершину. (Мы называем этот порядок ***последним вошел, первым вышел***, или ***LIFO*** — last in, first out.) При таком предположении список next на следующей итерации приобретает вид 3, 2, 9, и на следующей итерации цикла мы выбираем в качестве вершины *u* вершину 3. Мы удаляем ее из списка *next*, добавляем в конец линейного упорядочения, которое теперь имеет вид

"Трусы, шорты" и уменьшаем на единицу элементы *in-degree*[4] (с 2 до 1) и *in-degree*[5] (с 1 до 0). Вершину 5 (паховая защита) мы вносим в список *next*, который после этого имеет вид 5, 2, 9. На следующей итерации мы выбираем в качестве вершины *u* вершину 5, удаляем ее из списка *next*, добавляем в линейное упорядочение (которое теперь имеет вид "Трусы, шорты, паховая защита") и уменьшаем элемент *in-degree*[6] с 2 до 1. В этот раз новые вершины в список *next* не вносятся, так что на очередной итерации мы выбираем в качестве вершины *u* вершину 2 и т.д.



Чтобы проанализировать процедуру Topological-Sort, сначала надо понять, каким образом в компьютере представимы ориентированные графы и списки, такие как *next*. При представлении графов их цикличность или ацикличность не играет никакой роли.

Представление ориентированных графов

В компьютере ориентированный граф можно представить несколькими способами. Наше соглашение будет заключаться в том, что граф имеет n вершин и m ребер. Мы продолжаем считать, что каждая вершина имеет свой собственный номер от 1 до n, так что мы можем использовать вершину как индекс в массиве или даже как номер строки или столбца матрицы.

Пока что мы просто хотим знать, какие вершины и ребра имеются в наличии (в дальнейшем мы будем также с каждым ребром связывать некоторое числовое значение). Мы могли бы использовать ***матрицу смежности*** $n \times n$, в которой каждая строка и каждый столбец соответствует одной вершине, и запись в строке для вершины u и столбца для вершины v является либо 1, если в графе имеется ребро (u, v), либо 0, если граф не содержит такое ребро. Так как матрица смежности содержит n^2 записей, должно выполняться условие $m \le n^2$. В качестве альтернативы можно просто содержать список всех m ребер в графе в произвольном порядке. Гибридом между матрицей смежности и неупорядоченным списком является ***представление в виде списков смежности***, где n-элементный массив

индексируется вершинами, а запись массива для каждой вершины *u* представляет собой список всех вершин, смежных с ней. В сумме списки содержат *m* вершин, поскольку для каждого из *m* ребер имеется один элемент списка. Вот как выглядят матрица смежности и список смежности для ориентированного графа на с. 85.

Матрица смежности

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	1	0	0	0	0	0	0	0	0	0	0
3	0	0	0	1	1	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	1	0	0	0	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	1	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	0	1	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	1	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	1	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	1
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Списки смежности

1	3
2	4
3	4, 5
4	6
5	6
6	7, 11
7	8
8	13
9	10
10	11
11	12
12	13
13	14
14	Нет

Представления с помощью неупорядоченных списков ребер и списков смежности приводят к вопросу о выборе представления списка. Лучший способ представления списка зависит от того, операции какого вида должны будут выполняться над списком. Для неупорядоченных списков ребер и списков смежности мы знаем заранее количество ребер в этих списках, причем это количество не меняется со временем, так что можно хранить каждый список в массиве. Массив для хранения списка можно также использовать даже в случае, когда содержимое списка меняется с течением времени — до тех пор, пока известно максимальное количество элементов, которые могут находиться в списке в любой момент времени. Если нам не нужно вставлять элемент в средину списка или удалять элемент оттуда, представление списка массивом является столь же эффективным, как и любые иные средства.

Если нужны вставки в середину списка, можно воспользоваться ***связанным списком***, каждый элемент которого включает размещение его преемника в списке, что упрощает вставку нового элемента в список после данного элемента. Если необходимо также удаление из средины списка, то каждый элемент в связанном списке должен включать размещение его предшественника, что обеспечивает быстрое удаление элемента из списка.

Далее мы будем считать, что вставить элемент в связанный список или удалить его оттуда можно за константное время. Связанный список, который хранит только местоположение преемника, называется ***односвязным*** списком. Добавление в элемент ссылки на предшественника по списку делает список ***двусвязным***.

Время работы топологической сортировки

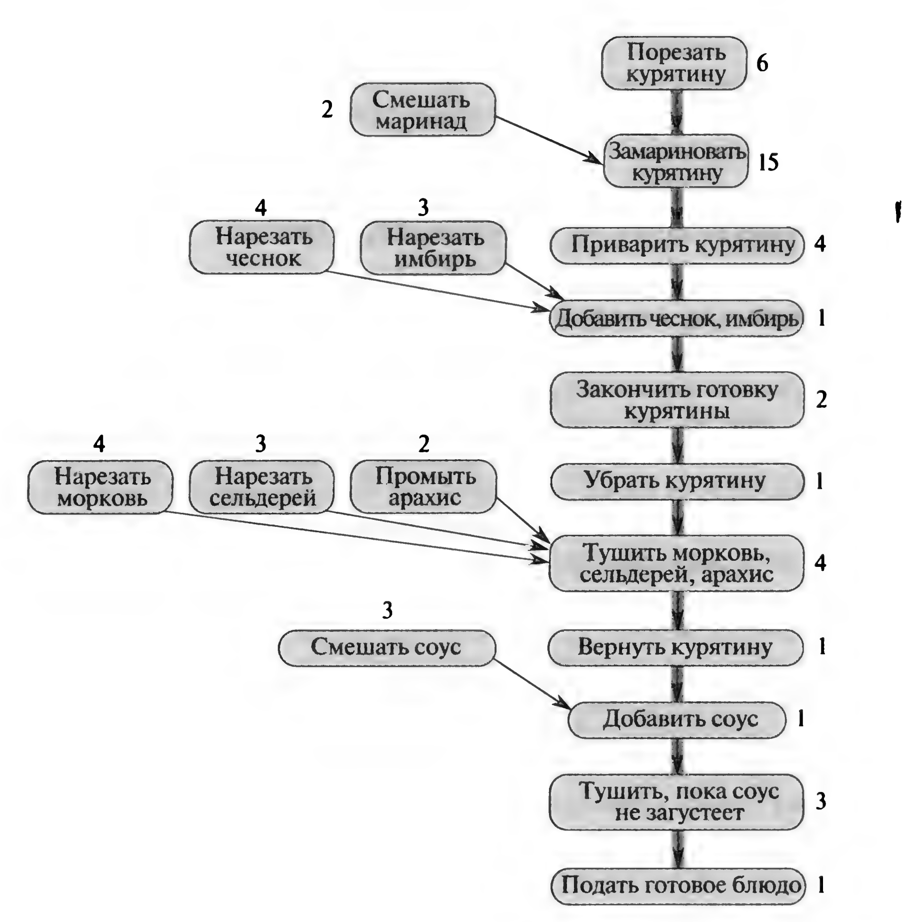
В предположении, что ориентированный ациклический граф использует представление в виде списков смежности, а список *next* — связанный список, можно показать, что процедура Topological-Sort выполняется за время $\Theta(n+m)$. Поскольку *next* — это связанный список, вставлять в него элементы и удалять их оттуда можно за константное время. Шаг 1 выполняется за константное время, а поскольку массив *in-degree* имеет n элементов, шаг 2 инициализирует массив нулями за время $\Theta(n)$. Шагу 3 требуется время $\Theta(n+m)$. Член $\Theta(n)$ на шаге 3 возникает из-за того, что внешний цикл просматривает каждую из n вершин, а член $\Theta(m)$ — потому что внутренний цикл на шаге 3A посещает каждое из m ребер ровно один раз за все итерации внешнего цикла. Шаг 4 выполняется за время $O(n)$, так как список *next* изначально содержит не более n вершин. Большая часть работы выполняется на шаге 5. Поскольку каждая вершина вносится в список *next* ровно один раз, выполняется n итераций главного цикла. Шаги 5A и 5B в каждой итерации выполняются за константное время. Подобно шагу 3A, цикл на шаге 5C всего выполняется m раз, по одному разу на ребро. Шаги 5Ci и 5Cii выполняются за константное время в каждой итерации, так что вместе все итерации шага 5C выполняются за время $\Theta(m)$, а следовательно, цикл на шаге 5 выполняется за время $\Theta(n+m)$. Конечно, шаг 6 выполняется за константное время, так что когда мы просуммируем все полученные времена, то найдем, что время работы топологической сортировки равно $\Theta(n+m)$.

Критический путь в диаграмме PERT

После напряженного рабочего дня я хочу расслабиться на кухне и приготовить на ужин курицу гунбао[1]. Я должен подготовить курятину, нарезать овощи, смешать маринад, сварить соус и приготовить блюдо. Так же, как и при одевании вратарской экипировки, одни шаги готовки должны быть выполнены раньше других, так что я могу воспользоваться ориентированным ациклическим графом для моделирования процедуры приготовления гунбао. Этот ориентированный ациклический граф показан ниже.

Рядом с каждой вершиной ориентированного ациклического графа приведено число, указывающее, сколько минут требуется на выполнение указанной в вершине задачи. Например, чтобы нарезать чеснок, мне надо четыре минуты (потому что сначала его надо почистить, а я всегда использую действительно *много* чеснока). Если вы просуммируете времена выполнения всех подзадач, то увидите, что при их последовательном выполнении я буду готовить гунбао целый час.

[1] *Или "кунг пао" — встречаются оба варианта этого названия китайского блюда на русском языке. — Примеч. пер.*



Но если мне будут помогать, то некоторые действия можно будет выполнять одновременно. Например, один человек может смешивать маринад, пока другой будет резать курятину. Имея достаточное количество помощников, большую кухню, много ножей, разделочных досок и тарелок, можно умудриться испортить гунбао за четыре часа. (Конечно, это шутка, но в каждой шутке есть доля истины...) Конечно же, я хотел сказать, что при этих условиях можно выполнять множество задач одновременно. Если на диаграмме между двумя задачами нет соединяющего их пути из стрелок, эти задачи можно выполнять одновременно, поручив их разным людям.

Как быстро можно приготовить гунбао при наличии неограниченных людских и кухонных ресурсов для одновременного выполнения нескольких задач? Наш ориентированный ациклический граф является примером ***диаграммы PERT*** (Project Evaluation and Review Technique — метод планирования и оценки затрат времени на проект с использованием сетевого графика). Время, требуемое для завершения всей работы, даже при максимально возможном распараллеливании задач определяется "критическим путем" диаграммы PERT. Чтобы понять, что такое критический путь, сначала надо понять, что такое путь вообще, после чего мы можем определить, что такое критический путь.

Путь в графе представляет собой последовательность вершин и ребер, которые позволят нам перейти от одной вершины к другой (или обратно). Мы говорим, что путь содержит вершины, лежащие на пути, и ребра, по которым он проходит. Например, один из путей в ориентированном ациклическом графе содержит вершины "Нарезать чеснок", "Добавить чеснок и имбирь", "Закончить готовку курятины" и "Убрать курятину" вместе ребрами, соединяющими эти вершины. Путь из вершины в нее же саму представляет собой ***цикл***, но, конечно же, ориентированные ациклические графы циклов не имеют.

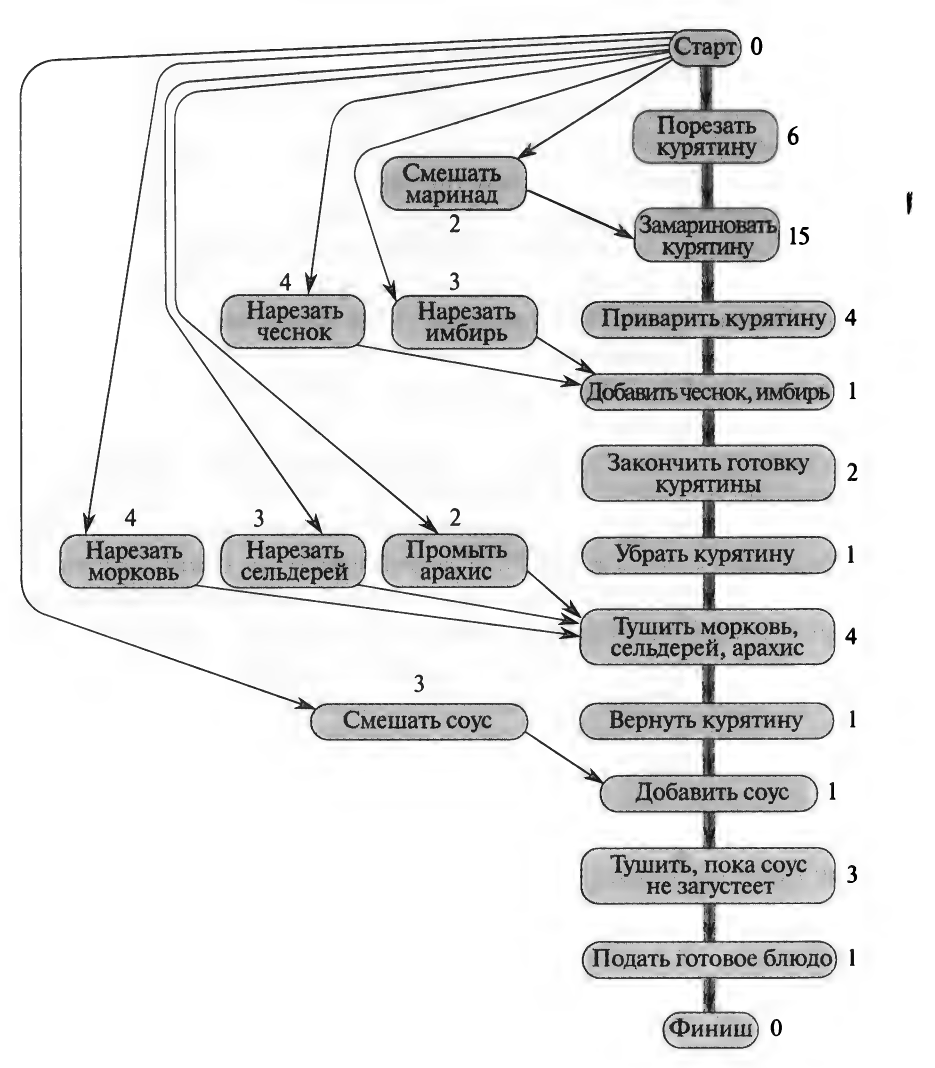
Критический путь в диаграмме PERT — это путь, сумма времен выполнения задач которого максимальна среди всех возможных путей. Сумма времен выполнения задач на критическом пути дает минимальное время выполнения всего задания, независимо от степени распараллеливания. Критический путь на диаграмме PERT для приготовления гунбао заштрихован. Если вы просуммируете времена выполнения задач вдоль критического пути, то увидите, что независимо от того, сколько у меня будет помощников, на приготовление блюда уйдет по крайней мере 39 минут[2].

В предположении, что времена выполнения всех задач являются положительными числами, критический путь диаграммы PERT должен начинаться с некоторой вершины с нулевой входящей степенью и заканчиваться в некоторой вершине с нулевой исходящей степенью. Вместо проверки путей между всеми парами вершин, в которых одна из них имеет нулевую входящую степень, а вторая — нулевую исходящую, можно просто добавить две фиктивные вершины — "старт" и "финиш", как показано на рисунке ниже. Поскольку это вершины фиктивные, мы назначаем каждой из них нулевое время выполнения. Добавим по ребру от вершины "старт" к каждой вершине с нулевой входящей степенью в диаграмме PERT, и ребро от каждой вершины нулевой исходящей степенью к вершине "финиш". Теперь нулевую входящую степень имеет только вершина "старт", и только вершина "финиш" имеет нулевую исходящую степень. Путь от старта до финиша с максимальной суммой времен выполнения задач (на рисунке он заштрихован) и является критическим путем в диаграмме PERT — конечно, после удаления фиктивных вершин "старт" и "финиш".

После того как мы добавили фиктивные вершины, мы находим критический путь как кратчайший путь от старта до финиша на основе указанных времен выполнения задач. Вы, наверное, думаете, что в предыдущем предложении я сделал ошибку, и критическому пути должен соответствовать самый длинный путь, а не кратчайший. Действительно, это так, но поскольку диаграмма PERT не содержит циклов, можно изменить времена выполнения задач так, чтобы критический путь соответствовал кратчайшему пути. В частности, можно изменить знак времени выполнения каждой задачи и найти путь от старта до финиша с *минимальным* суммарным временем выполнения задач.

---

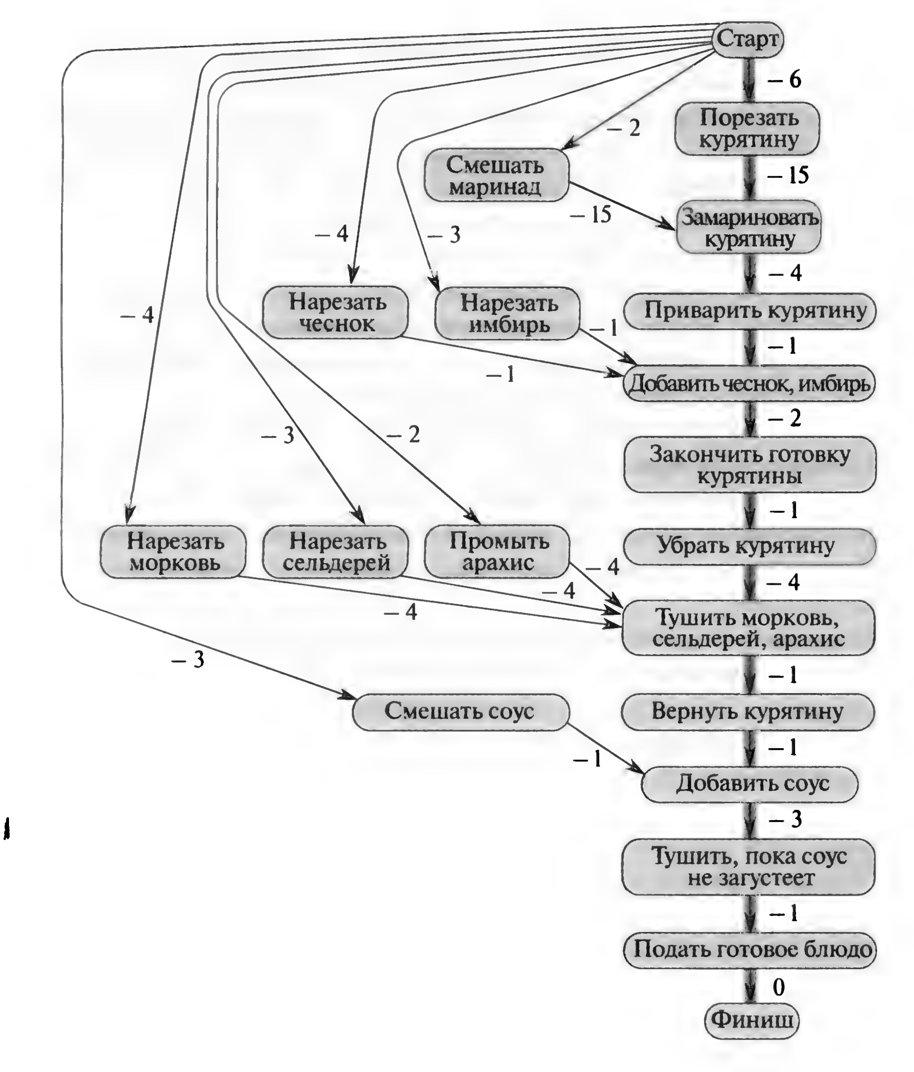
[2] *Вас интересует, почему в китайских ресторанах вы обычно ждете гораздо меньшее время? Дело в том, что они могут ряд ингредиентов подготавливать заранее, в расчете на возможных посетителей; может быть, их печи в состоянии готовить быстрее моей домашней. А может, видя, что перед ними не особо искушенный гурман, они просто быстренько разогревают в микроволновке блюдо, приготовленное еще вчера...*



Почему мы делаем это — меняем знак времени выполнения задач и ищем путь с минимальным суммарным временем? Потому что решение этой задачи является частным случаем задачи поиска кратчайших путей, для решения которой разработана масса алгоритмов. Однако когда мы говорим о кратчайшем пути, значения, которые определяют длину пути, связаны с его ребрами, а не вершинами. Такое значение, связанное с ребром, называется его ***весом***. Ориентированный граф, в котором ребра имеют веса, называется ***взвешенным ориентированным графом***. "Вес" является обобщенным термином для значений, связанных с ребрами. Если взвешенный ориентированный граф представляет дорожную сеть, каждое ребро в нем представляет одно направление дороги между двух перекрестков, а вес ребра может представлять длину дороги, время, необходимое для поездки, или для платных дорог — сумму, которую следует заплатить, чтобы использовать эту дорогу. ***Вес пути*** представляет собой сумму весов ребер, принадлежащих этому пути,

так что если веса ребер указывают расстояние между перекрестками, то вес пути представляет собой общую длину маршрута. ***Кратчайший путь*** из вершины u в вершину v представляет собой путь, сумма весов ребер которого минимальна среди всех возможных путей из вершины u в вершину v. Кратчайшие пути не обязательно единственные, и ориентированный граф может иметь несколько путей, вес которых достигает одного и того же минимального значения.

Чтобы преобразовать диаграмму PERT с отрицательными временами решения задач во взвешенный ориентированный граф, мы переносим время работы каждой задачи с обратным знаком в каждое из входящих в соответствующую вершину ребер. То есть если вершина v имеет (не отрицательное) время выполнения задачи t, то мы устанавливаем вес каждого входящего в нее ребра (u, v) равным $-t$. Вот ориентированный ациклический граф, который получается при таком действии для диаграммы приготовления гунбао.



Теперь мы просто должны найти в этом ориентированном ациклическом графе кратчайший путь (на рисунке он заштрихован) от старта до финиша, основываясь на весах ребер. Критический путь в исходной диаграмме PERT будет соответствовать вершинам на найденном нами кратчайшем пути с удаленными вершинами "старт" и "финиш". Так что давайте познакомимся с тем, как можно найти кратчайший путь в ориентированном ациклическом графе.

Кратчайший путь в ориентированном ациклическом графе

В изучении поиска кратчайшего пути в ориентированном ациклическом графе имеется еще одно преимущество: тем самым мы заложим основы для поиска кратчайшего пути в произвольных ориентированных графах, которые могут содержать циклы. Эту более общую задачу мы рассмотрим в главе 6, "Кратчайшие пути". Как и в случае топологической сортировки ориентированного ациклического графа, мы предполагаем, что он хранится в виде списков смежности и что каждое ребро (u, v) имеет свой вес $weight(u, v)$.

В ориентированном ациклическом графе, который получается из диаграммы PERT, нам надо найти кратчайший путь от ***исходной вершины***, которую мы называем "стартом", к ***целевой вершине***, "финишу". Здесь мы будем решать более общую задачу поиска ***кратчайших путей из одной вершины***, в которой будем искать кратчайшие пути из исходной вершины ко *всем* другим вершинам. Примем соглашение, по которому будем именовать исходную вершину s, и при этом нам нужно вычислить для каждой вершины v две характеристики: во-первых, вес кратчайшего пути от s до v, который мы будем обозначать как $sp(s, v)$, а во-вторых — вершину, являющуюся предшественником v на кратчайшем пути от s до v, т.е. вершину u, такую, что кратчайший путь от s до v является путем от s до u, к которому добавлено одно ребро (u, v). Пронумеруем n вершин числами от 1 до n, так что наши алгоритмы для поиска кратчайшего пути здесь и в главе 6, "Кратчайшие пути", могут хранить указанные результаты в массивах $shortest[1..n]$ и $pred[1..n]$ соответственно. В процессе выполнения алгоритмов промежуточные значения в массивах $shortest[1..n]$ и $pred[1..n]$ могут не быть верными конечными значениями, но по окончании работы они становятся таковыми.

Нам нужно уметь обрабатывать несколько могущих возникнуть при решении поставленной задачи случаев. Во-первых, что если пути от s к v вообще нет? Тогда мы определяем $sp(s, v) = \infty$, так что элемент массива $shortest[v]$ должен получить значение ∞. Поскольку вершина v при этом не должна иметь предшественника на кратчайшем пути от s, элемент $pred[v]$ также должен иметь специальное значение NULL. Кроме того, все кратчайшие пути от s начинаются с s, а потому s также не имеет предшественника; таким образом, $pred[s]$ также должен иметь значение NULL. Другой случай возникает только в графах, у которых имеются циклы и отрицательные веса ребер: что делать, если суммарный вес цикла является отрицательным? В этом случае можно делать круги по циклу, с каждым кругом получая все меньшее и меньшее значение пути. Получается, что если можно достичь от

вершины s цикла с отрицательным весом, а от него достичь вершину v, то вес $sp(s,v)$ не определен. Сейчас, однако, мы имеем дело только с ациклическими графами, в которых нет циклов, и еще меньше циклов с отрицательным весом, так что беспокоиться об этом случае еще рано.

Чтобы вычислить кратчайший путь от исходной вершины s, мы начинаем с $shortest[s] = 0$ (так как нам не нужно никуда идти, чтобы достичь из вершины ее же), и $shortest[v] = \infty$ для всех других вершин v (так как мы не знаем заранее, до каких вершин можно добраться от s). По той же причине для всех вершин v изначально $pred[v] = \text{NULL}$. Затем мы выполняем ряд ***шагов ослабления*** к ребрам графа.

Процедура RELAX(u,v)

Вход: u, v: вершины, такие, что в графе имеется ребро (u,v).

Результат: значение $shortest[v]$ может уменьшиться, и в этом случае $pred[v]$ принимает значение u.

1. Если $shortest[u] + weight(u,v) < shortest[v]$, установить $shortest[v]$ равным $shortest[u] + weight(u,v)$, а $pred[v]$ равным u.

Вызов RELAX(u,v) определяет, можно ли улучшить текущей кратчайший путь от s к v, используя в качестве последнего ребра (u,v). Мы сравниваем вес текущего кратчайшего пути к u плюс вес ребра (u,v) с весом текущего кратчайший пути к v. Если лучшим решением оказывается использовать ребро (u,v), соответствующим образом обновляются элементы массивов $shortest[v]$ и $pred[v]$.

Если мы последовательно ослабляем ребра вдоль кратчайшего пути, то получим корректный результат. Вы можете спросить, как мы можем быть уверены в ослаблении ребер вдоль кратчайшего пути, если мы даже не знаем, каков этот путь (в конце концов, именно это мы и пытаемся выяснить)? Оказывается, что в случае ориентированного ациклического графа это очень легко. Мы собираемся ослабить все ребра ориентированного ациклического графа, и ребра каждого кратчайшего пути при проходе и ослаблении всех ребер окажутся размещенными среди них в верном порядке.

Вот более точное описание работы ослабления ребер вдоль кратчайших путей, которое применимо к любому ориентированному графу, независимо от наличия в нем циклов.

Начнем с $shortest[u] = \infty$ и $pred[u] = \text{NULL}$ для всех вершин, за исключением исходной, для которой $shortest[s] = 0$.

Затем ослабляем ребра вдоль кратчайшего пути от s до любой вершины v *по порядку, начиная от ребра, исходящего из s, и заканчивая ребром, входящим в v*. Ослабления других ребер могут свободно перемежаться с ослаблениями вдоль кратчайшего пути, но изменять какие-либо значения массивов $shortest$ и $pred$ могут только ослабления.

После ослабления ребер значения элементов массивов *shortest* и *pred* для вершины v являются правильными: $shortest[v] = sp(s,v)$, а $pred[v]$ представляет собой вершину, предшествующую v на некотором кратчайшем пути от s.

Легко понять, почему работает ослабление ребер по порядку вдоль кратчайшего пути. Предположим, что кратчайший путь от s к v проходит через вершины $s, v_1, v_2, v_3, \ldots, v_k, v$ в указанном порядке. После ослабления ребра (s, v_1) значение $shortest[v_1]$ должно представлять собой корректный вес кратчайшего пути до v_1, а элемент $pred[v_1]$ должен быть равен s. После ослабления ребра (v_1, v_2) должны получить верные значения $shortest[v_2]$ и $pred[v_2]$. И так далее до ослабления ребра (v_k, v), после которого правильные значения получают $shortest[v]$ и $pred[v]$.

Это хорошая новость. В ориентированном ациклическом графе легко ослабить каждое ребро ровно один раз по порядку вдоль *каждого* кратчайшего пути. Каким образом? Сначала выполним топологическую сортировку ориентированного ациклического графа. Затем рассмотрим каждую вершину в линейном порядке, полученном путем топологической сортировки, и ослабим все ребра, покидающие вершины. Поскольку каждое ребро должно покидать вершину, идущую в линейном порядке ранее, и входить в вершину, более позднюю в этом порядке, каждый путь в ориентированном ациклическом графе должен посещать вершины в порядке, согласующемся с линейным порядком, получаемым топологической сортировкой.

Процедура DAG-SHORTEST-PATHS(G,s)

Вход:

- G: взвешенный ориентированный ациклический граф, содержащий множество V из n вершин и множество E из m ориентированных ребер.
- s: исходная вершина из V.

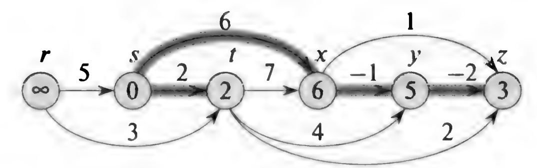
Результат: для каждой вершины v из V, не являющейся исходной, значение $shortest[v]$ равно весу $sp(s,v)$ кратчайшего пути от s к v, а элемент $pred[v]$ представляет собой вершину, предшествующую v на некотором кратчайшем пути. Для исходной вершины s $shortest[s] = 0$ и $pred[s] =$ NULL. Если пути из s к v нет, то $shortest[v] = \infty$ и $pred[v] =$ NULL.

1. Вызвать процедуру TOPOLOGICAL-SORT(G) и определить линейный порядок l вершин графа, возвращаемый топологической сортировкой.
2. Для каждой вершины v, отличной от s, установить $shortest[v] = \infty$; установить $shortest[s] = 0$ и для всех вершин v графа установить $pred[v] =$ NULL.
3. Для каждой вершины u, взятой в линейном порядке l:

   A. Для каждой вершины v, смежной с u:

      i. Вызвать RELAX(u,v).

На приведенном далее рисунке показан ориентированный ациклический граф, рядом с каждым ребром которого указан его вес. Значения *кратчайших* путей от вершины s, вычисленные с помощью вызова DAG-SHORTEST-PATHS, указаны внутри вершин, а заштрихованные ребра показывают значения *pred*. Вершины располагаются слева направо в линейном порядке, полученном с помощью топологической сортировки, так что все ребра направлены слева направо. Если ребро (u,v) заштриховано, то $pred[v]=u$ и $shortest[v]=shortest[u]+weight(u,v)$; например, поскольку ребро (x,y) заштриховано, $pred[y]=x\,pred[y]=x$, а $shortest[y]$ (которое равно 5) представляет собой $shortest[x]$ (равное 6) плюс $weight(x,y)$ (равно -1). Пути из s к r нет, так что $shortest[r]=\infty$ и $pred[r]=$ NULL (нет заштрихованных ребер, входящих в r).



Первая итерация цикла на шаге 3 ослабляет ребра (r,s) и (r,t), покидающие r, но поскольку $shortest[r]=\infty$, это ослабление ничего не меняет. На следующей итерации цикла ослабляются ребра (s,t) и (s,x), покидающие s, что приводит к тому, что $shortest[t]$ становится равным 2, $shortest[x]$ — равным 6, а оба элемента — $pred[t]$ и $pred[x]$ — устанавливаются равными s. Очередная итерация ослабляет ребра (t,x), (t,y) и (t,z), покидающие t. Значение $shortest[x]$ не изменяется, поскольку $shortest[t]+weight(t,x)$, равное $2+7=9$, превышает значение $shortest[x]$, равное 6. Однако $shortest[y]$ становится равным 6, $shortest[z]$ получает значение 4, а оба элемента — $pred[y]$ и $pred[z]$ — устанавливаются равными t. Следующая итерация ослабляет ребра (x,y) и (x,z), покидающие вершину x, после чего $shortest[y]$ становится равным 5, а $pred[y]$ — вершине x; $shortest[z]$ и $pred[z]$ остаются неизменными. Последняя итерация ослабляет ребро (y,z), в результате чего $shortest[z]$ получает значение 3, а $pred[z]$ — y.

Можно легко увидеть, что поиск кратчайшего пути в ориентированном ациклическом графе выполняется за время $\Theta(n+m)$. Как мы знаем, шаг 1 выполняется за время $\Theta(n+m)$, а шаг 2, конечно же, инициализирует по два значения для каждой вершины за время $\Theta(n)$. Как мы уже видели ранее, внешний цикл на шаге 3 рассматривает каждую вершину только один раз, а внутренний цикл на шаге 3А делает то же с каждым ребром — рассматривает его ровно один раз за все итерации. Поскольку каждый вызов процедуры RELAX на шаге 3Ai выполняется за константное время, время работы шага 3 составляет $\Theta(n+m)$. Суммирование времен работы для всех шагов дает нам общее время работы процедуры, равное $\Theta(n+m)$.

Возвращаясь к диаграмме PERT с n вершинами и m ребрами, легко увидеть, что поиск критического пути выполняется за время $\Theta(n+m)$. Мы добавили к диаграмме две вершины (старт и финиш), а также не более m ребер, покидающих старт, и m ребер, входящих в финиш, т.е. в общей сложности в получившемся ориентированном ациклическом графе имеется не более $3m$ ребер. Изменение знака весов и перенос их из вершин в ребра

выполняется за время $\Theta(m)$, а следующий за этим поиск кратчайшего пути в получившемся ориентированном ациклическом графе выполняется за время $\Theta(n+m)$.

Дальнейшее чтение

Глава 22 CLRS [4] содержит другой алгоритм топологической сортировки ориентированного ациклического графа, отличный от представленного в данной главе (этот алгоритм взят из тома 1 *Искусства программирования* Кнута (Donald Knuth) [10]). Метод в CLRS немного проще, но менее интуитивно понятный, чем приведенный в этой главе, и опирается на методику обхода вершин графа, известную как “поиск в глубину”. Алгоритм поиска кратчайших путей из одной вершины приведен в главе 24 CLRS.

О диаграммах PERT, используемых с 1950-х годов, можно прочесть в любой из множества книг об управлении проектами.

6... Кратчайшие пути

В главе 5, "Ориентированные ациклические графы", вы познакомились с одним из способов поиска кратчайших путей из одной вершины в ориентированном ациклическом графе.

Однако большинство графов, моделирующих явления реальной жизни, содержат циклы. Например, в графе, моделирующем дорожную сеть, каждая вершина представляет перекресток, а каждое ориентированное ребро — дорогу, по которой вы можете двигаться в одном направлении между перекрестками (дороге с двусторонним движением соответствуют два отдельных ребра, идущие в противоположных направлениях). Такие графы должны содержать циклы, иначе после того, как вы проехали перекресток, вы уже никогда не могли бы к нему вернуться. Таким образом, когда ваш GPS вычисляет кратчайший или скорейший маршрут к месту назначения, граф, с которым он работает, содержит множество циклов.

Когда ваш GPS ищет самый быстрый маршрут от вашего текущего местоположения в указанное место назначения, он решает задачу ***поиска кратчайшего пути между парой вершин***. Чтобы ее решить, он, вероятно, использует алгоритм, который находит все кратчайшие пути от одной вершины, но затем GPS уделяет внимание только тому из кратчайших путей, который приводит к требуемому месту назначения.

Ваш GPS работает со взвешенным ориентированным графом, веса ребер которого представляют собой либо расстояние, либо время в пути. Поскольку нельзя ни проехать отрицательное расстояние, ни прибыть в место назначения до того, как вы отправились в дорогу, все веса ребер в графе, с которым работает ваш GPS, являются положительными. Я допускаю, что некоторые из них могут оказаться равными нулю по каким-то непонятным причинам, так что давайте просто говорить о неотрицательных весах ребер. Когда все веса ребер неотрицательные, нам незачем беспокоиться о циклах с отрицательным весом, так что все кратчайшие пути являются точно определенными.

Имеется масса других примеров из жизни, в которых ситуацию можно описать ориентированными графами с неотрицательными весами ребер. А есть ли явления реального мира, при описании которых получается граф с отрицательными весами ребер? Да, например, в случае обмена валют возможны ситуации, описываемые графами, в которых есть ребра с отрицательным весом, и даже циклы с отрицательным весом.

Переходя к алгоритмам, мы сначала изучим алгоритм Дейкстры поиска кратчайших путей от одной вершины до всех остальных вершин графа. Алгоритм Дейкстры работает с графами, которые имеют два важных отличия от графов, с которыми мы встречались в главе 5, "Ориентированные ациклические графы": все веса ребер должны быть неотрицательными, и граф может содержать циклы. Это ключевой момент в поиске маршрутов вашим GPS. Мы также рассмотрим несколько вариантов реализации алгоритма Дейкстры. Затем мы познакомимся с алгоритмом Беллмана–Форда, удивительно простым методом поиска кратчайших путей из одной вершины даже при наличии ребер с отрицательным ве-

сом. Алгоритм Беллмана–Форда можно использовать для определения, содержит ли граф цикл с отрицательным весом, и если содержит, то он позволяет указать вершины и ребра, входящие в такой цикл. И алгоритм Дейкстры, и алгоритм Беллмана–Форда датируются концом 1950-х годов, так что они выдержали испытание временем. Рассмотрение темы мы завершим алгоритмом Флойда–Уоршелла для поиска кратчайших путей между всеми парами вершин графа.

Так же, как мы делали в главе 5, "Ориентированные ациклические графы", для поиска кратчайшего пути в ориентированном ациклическом графе, мы предполагаем, что у нас заданы исходная вершина s (источник) и вес $weight(u,v)$ каждого ребра (u,v), и хотим вычислить для каждой вершины v вес кратчайшего пути $sp(s,v)$ от s к v, а также вершину, предшествующую v на некотором кратчайшем пути от s. Мы будем хранить результаты в элементах массивов $shortest[v]$ и $pred[v]$ соответственно.

Алгоритм Дейкстры

Мне нравится представлять алгоритм Дейкстры[1] как имитацию действий бегунов, бегущих по графу.

В идеале такая модель работает так, как показано далее, хотя вы увидите некоторые ее отличия от алгоритма Дейкстры. Она начинается с отправки бегунов из исходной вершины во все соседние. Как только бегун впервые достигает любой вершины, из нее тут же выбегают бегуны во все соседние вершины. Взгляните на часть (а) рисунка.

На ней показан ориентированный граф с исходной вершиной s и весами всех ребер. Рассматривайте вес ребра как количество минут, требующееся бегуну для того, чтобы преодолеть это ребро.

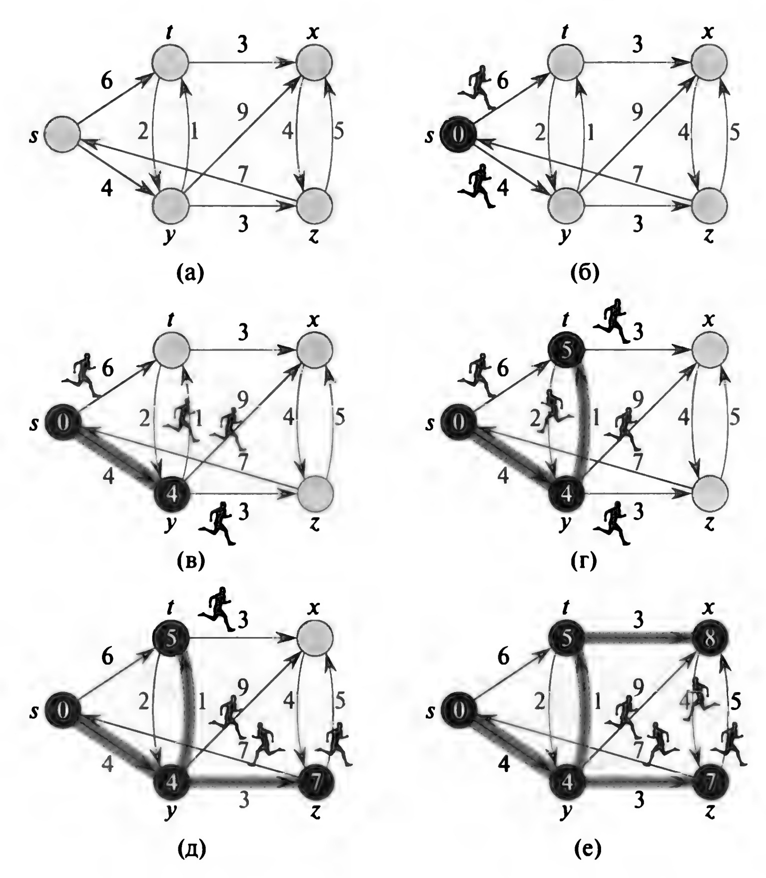
В части (б) показано начало процесса моделирования — нулевой момент времени. В этот момент, показанный внутри вершины s, бегуны покидают s и направляются к двум смежным с ней вершинам, t и y. Затенение вершины s означает, что мы знаем, что $shortest[s]=0$.

Четыре минуты спустя, в момент времени 4, бегун прибывает в вершину y (что показано в части (в)). Поскольку этот бегун первый прибывший в вершину y, мы знаем, что $shortest[y]=4$, и вершина y на рисунке затенена. Заштрихованное ребро (s,y) указывает, что первый бегун прибыл в вершину y из вершины s, так что $pred[y]=s$. В момент 4 бегун из вершин s к вершине t все еще находится в пути, и в этот же момент времени 4 вершину y покидают бегуны, направляющиеся к вершинам t, x и z.

Следующее событие, отображенное в части (г), происходит одну минуту спустя, в момент времени 5, когда бегун из вершины y прибывает в вершину t. Бегун из s в t пока что не успевает. Поскольку первый бегун прибыл в вершину t из вершины y в момент времени 5, мы устанавливаем $shortest[t]=5$ и $pred[t]=y$ (что указывается штриховкой ребра (y,t)). Из вершины t выбегают бегуны, которые теперь направляются к вершинам x и y.

Наконец в момент времени 6, бегун из вершины s добегает до вершины t, но так как бегун из вершины y уже побывал там минутой ранее, усилия бегуна из s к t были напрасны.

[1] *Назван по имени Эдсгера Дейкстры (Edsger Dijkstra), предложившего этот алгоритм в 1959 году.*



В момент времени 7, показанный в части (д), два бегуна прибывают в пункт назначения. Бегун из вершины *t* прибегает в вершину *y*, но там уже побывал бегун из *s* в момент времени 4, так что о бегуне из *t* в *y* можно просто забыть. В этот же момент времени бегун из *y* прибывает в вершину *z*. Мы устанавливаем $shortest[z]=7$ и $pred[z]=y$, и бегуны выбегают из вершины *z*, направляясь к вершинам *s* и *x*.

Следующее событие происходит в момент времени 8, показанный в части (е), когда бегун из вершины *t* прибывает в вершину x. Мы устанавливаем $shortest[x]=8$ и $pred[x]=t$, и бегун покидает вершину *x*, направляясь в вершину *z*.

Теперь, когда бегуны побывали в каждой вершине, моделирование можно остановить. Конечно, некоторые бегуны еще в пути, но их прибытие будет в любом случае позже первых бегунов, посетивших эти вершины первыми. Как только в вершину прибывает первый бегун, момент его прибытия определяет кратчайший путь от вершины *s*, а значение элемента массива *pred* определяет предшественника данной вершины на кратчайшем пути из *s*.

Выше описано, как моделирование должно выполняться в идеале. Оно основано на времени прохода бегуна по ребру, равному весу ребра. Алгоритм Дейкстры работает немного иначе. Он рассматривает все ребра как одинаковые, так что, когда он рассматривает

ребра, покидающие вершину, он обрабатывает все смежные вершины одновременно, в произвольном порядке. Например, когда алгоритм Дейкстры обрабатывает ребра, покидающие вершину s на рисунке на с. 99, он объявляет, что $shortest[y]=4$, $shortest[t]=6$, а $pred[y]$ и $pred[t]$ оба равны s — *пока что*. Когда алгоритм Дейкстры позже рассмотрит ребро (y,t), это приведет к снижению веса кратчайшего пути к вершине t, найденного до этого момента, так что $shortest[t]$ станет равным 5 вместо 6, а $pred[t]$ вместо s примет значение y.

Алгоритм Дейкстры работает путем вызова процедуры Relax (см. с. 93) по одному разу для каждого ребра. Ослабление ребра (u,v) соответствует бегуну, бегущему из вершины u в вершину v. Алгоритм поддерживает множество Q вершин, для которых окончательные значения *shortest* и *pred* пока неизвестны; все вершины, *не* входящие в Q, уже получили окончательные значения *shortest* и *pred*. После инициализации $shortest[s]=0$, а для всех остальных вершин $shortest[v]=\infty$; значение $pred[v]=$ NULL для всех вершин. Алгоритм многократно выполняет следующие действия — находит в множестве Q вершину u с наименьшим значением *shortest*, удаляет ее из Q и ослабляет все ребра, выходящие из u.

Процедура Dijkstra(G,s)

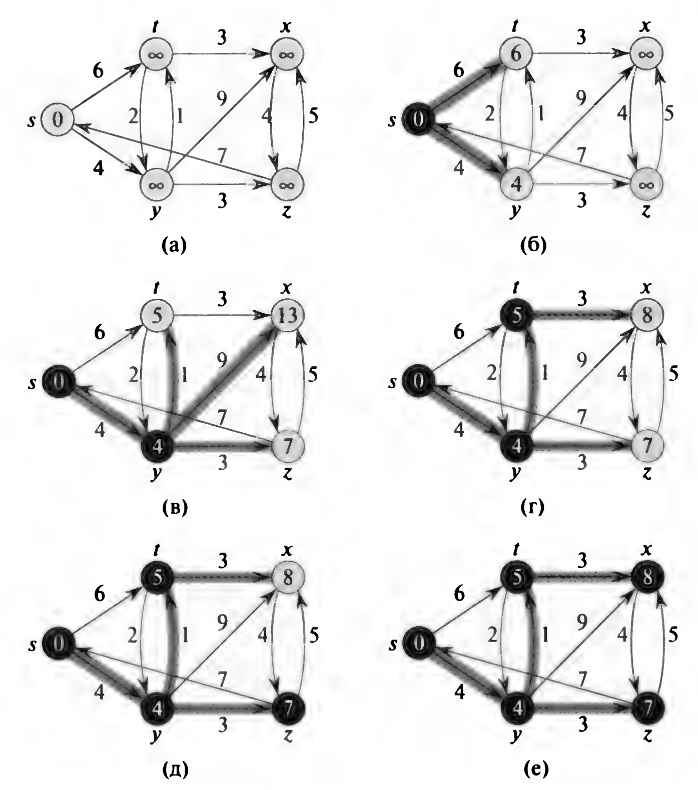
Вход:

- G: ориентированный граф, содержащий множество V из n вершин и множество E из m ориентированных ребер с неотрицательными весами.
- s: исходная вершина из множества V.

Результат: для каждой вершины v из V, не являющейся исходной, $shortest[v]$ содержит вес $sp(s,v)$ кратчайшего пути из s в v, а $pred[v]$ представляет собой вершину, предшествующую v на некотором кратчайшем пути. Для исходной вершины s $shortest[s]=0$ и $pred[s]=$ NULL. Если пути из s в v нет, то $shortest[v]=\infty$ и $pred[v]=$ NULL. (Результат тот же, что и у процедуры Dag-Shortest-Paths на с. 94.)

1. Установить $shortest[v]=\infty$ для всех вершин v, за исключением s; $shortest[s]=0$, и для всех вершин v — $pred[v]=$ NULL.
2. Внести все вершины в множество Q.
3. Пока множество Q не пустое, выполнять следующие действия.
   A. Найти в множестве Q вершину u с наименьшим значением *shortest* и удалить ее из Q.
   B. Для каждой вершины v, смежной с u:
      i. Вызвать Relax(u,v).

В каждой части следующего рисунка показаны значения *shortest* (приведены в вершинах графа), значения *pred* (обозначены заштрихованными ребрами) и множество Q (*не* затененные вершины) перед каждой итерацией цикла на шаге 3 алгоритма.



Затеняемая на каждой итерации вершина — это и есть выбранная вершина *u* на шаге 3А. В модели с бегунами после того, как вершина получает значения *shortest* и *pred*, они не могут быть впоследствии изменены, но в данном случае вершина может получить новые значения *shortest* и *pred* в результате ослабления некоторых других ребер. Например, после ослабления ребра (y,x) в части (в) рисунка значение $shortest[x]$ уменьшается от ∞ до 13, а $pred[x]$ становится равным *y*. Очередная итерация цикла на шаге 3 (часть (г)) ослабляет ребро (t,x), и $shortest[x]$ уменьшается еще больше — до 8. При этом $pred[x]$ становится равным *t*. На следующей итерации (часть (д)) ослабляется ребро (z,x), но в этот раз значение $shortest[x]$ не изменяется, поскольку его значение 8 оказывается меньшим, чем $shortest[z] + weight(z,x)$, равное 12.

Алгоритм Дейкстры поддерживает следующий инвариант цикла.

> В начале каждой итерации цикла на шаге 3 $shortest[v] = sp(s,v)$ для каждой вершины *v*, не входящей в *Q*. То есть для каждой вершины *v*, не входящей в *Q*, значение $shortest[v]$ представляет собой вес кратчайшего пути от *s* до *v*.

Вот упрощенная версия обоснования этого инварианта цикла (формальное доказательство немного сложнее). Изначально все вершины входят в множество *Q*, так что инвариант цикла не применяется к вершинам до входа в первую итерацию цикла на шаге 3. Предположим, что при входе в этот цикл все вершины, не входящие в множество *Q*, имеют

в *shortest* корректные значения весов кратчайших путей. Тогда каждое ребро, покидающее эти вершины, было ослаблено при некотором выполнении шага 3Bi. Рассмотрим вершину u из Q с наименьшим значением *shortest*. Ее значение *shortest* никогда не сможет уменьшиться. Почему? Потому что единственные ребра, которые могут быть ослаблены, — это ребра, выходящие из вершин в Q, а каждая вершина в Q имеет значение *shortest*, не меньшее, чем $shortest[u]$. Так как все веса ребер неотрицательные, для каждой вершины v в Q должно выполняться $shortest[u] \le shortest[v] + weight(v,u)$, так что ни одно будущее ослабление не уменьшит значение $shortest[u]$. Таким образом, $shortest[u]$ имеет наименьшее возможное значение, и мы можем удалить u из Q и выполнить ослабление всех выходящих из u ребер. По завершении цикла на шаге 3 множество Q становится пустым, так что у всех вершин в их значениях *shortest* оказываются корректные веса кратчайших путей.

Мы можем начать анализ времени работы процедуры Dijkstra, но, чтобы проанализировать ее в полном объеме, сначала необходимо согласовать некоторые детали ее реализации. Вспомним, что в главе 5, "Ориентированные ациклические графы", мы обозначали количество вершин n и количество ребер m, и при этом $m \le n^2$. Мы знаем, что шаг 1 выполняется за время $\Theta(n)$. Мы также знаем, что цикл на шаге 3 выполняет итерации ровно n раз, потому что множество Q изначально содержит все n вершин, а каждая итерация цикла удаляет по одной вершине из Q, причем вершины обратно в Q никогда не добавляются. Цикл на шаге 3A обрабатывает каждую вершину и каждое ребро ровно один раз в течение выполнения алгоритма (с такой же идеей мы сталкивались при работе с процедурами Topological-Sort и Dag-Shortest-Paths в главе 5, "Ориентированные ациклические графы").

Что же осталось проанализировать? Нам нужно понять, сколько времени потребуется, чтобы поместить все n вершин в множество Q (шаг 2), за какое время можно найти вершину в Q с наименьшим значением *shortest* и удалить эту вершину из Q (шаг 3A) и какие действия надо выполнить при изменении значений *shortest* и *pred* вследствие вызова процедуры Relax. Дадим этим операциям имена.

- Insert(Q,v) вставляет вершину v в множество Q. (Алгоритм Дейкстры вызывает процедуру Insert n раз.)
- Extract-Min(Q) удаляет из Q вершину с минимальным значением *shortest* и возвращает эту вершину вызывающей процедуре. (Алгоритм Дейкстры вызывает процедуру Extract-Min n раз.)
- Decrease-Key(Q,v) выполняет все необходимые действия над Q, чтобы записать, что значение $shortest[v]$ уменьшилось при вызове процедуры Relax. (Алгоритм Дейкстры вызывает процедуру Decrease-Key до m раз.)

Взятые вместе, эти три операции определяют ***очередь с приоритетами***.

Описания очереди с приоритетами говорят только о том, *что* делают ее операции, но не *как* они это делают. В разработке программного обеспечения отделение того, *что* делают операции, от того, *как* они это делают, известно как ***абстракция***. Мы называем набор операций, определяемых тем, *что*, но не *как* они делают, ***абстрактным типом данных***, или АТД, очередь с приоритетами является АТД.

Реализовать очереди с приоритетами — то самое *как* — можно с помощью одной из нескольких структур данных. ***Структура данных*** представляет собой конкретный способ хранения и доступа к данным на компьютере, например массив. В случае очереди с приоритетами мы рассмотрим три различные структуры данных, позволяющие реализовать указанные операции. Разработчики программного обеспечения должны быть способны использовать любую структуру данных, которая реализует операции АТД. Но это совсем не так просто, когда мы говорим об алгоритмах. Дело в том, что для различных структур данных способ реализации одной и той же операции может привести к разному времени работы. Три различные структуры данных, о которых сказано выше, действительно дают различные времена работы алгоритма Дейкстры.

Переписанная версия процедуры DIJKSTRA, явно вызывающая операции очереди с приоритетами, приведена ниже. Рассмотрим три структуры данных для реализации приоритетных операций очереди с приоритетами и их влияние на время работы алгоритма Дейкстры.

Процедура DIJKSTRA(G,s)

Вход и результат: Те же, что и ранее.

1. Установить $shortest[v] = \infty$ для всех вершин v, за исключением s, $shortest[s] = 0$, и для всех вершин v — $pred[v] =$ NULL.
2. Сделать Q пустой очередью с приоритетами.
3. Для каждой вершины v:

   A. Вызвать INSERT(Q,v).
4. Пока очередь Q не пуста, выполнять следующие действия.

   A. Вызвать EXTRACT-MIN(Q) и присвоить u возвращенную вершину.

   B. Для каждой вершины v, смежной с u:

      i. Вызвать RELAX(u,v).

      ii. Если вызов RELAX(u,v) уменьшает значение $shortest[v]$, вызвать DECREASE-KEY(Q,v).

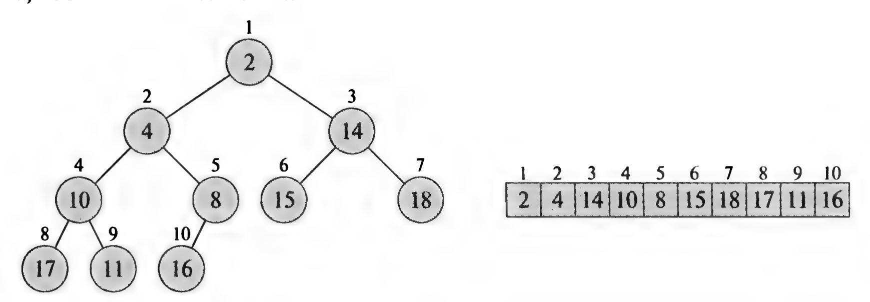
Простая реализация с помощью массива

Самый простой способ реализации операций очереди с приоритетами — хранение вершин в массиве с n элементами. Если в текущий момент очередь с приоритетами содержит k вершин, то они находятся в первых k позициях массива в произвольном порядке. Наряду с массивом необходимо поддерживать счетчик, указывающий, сколько вершин в настоящее время находится в массиве. Операция INSERT реализуется легко: надо просто добавить вершину в первую неиспользуемую позицию в массиве и выполнить приращение значения счетчика. Операция DECREASE-KEY еще проще: делать не надо вообще ничего! Обе эти операции выполняются за константное время. Однако операция EXTRACT-MIN выполняется за время $O(n)$, поскольку мы должны просмотреть все вершины, имеющиеся

в текущий момент в массиве, чтобы найти вершину с наименьшим значением *shortest*. Как только мы обнаружим эту вершину, удалить ее будет достаточно легко: нужно просто переместить вершину из последней позиции в позицию удаляемой и соответственно уменьшить значение счетчика. Время выполнения n вызовов процедуры EXTRACT-MIN составляет $O(n^2)$. Хотя вызов процедуры RELAX выполняется за время $O(m)$, вспомните, что $m \leq n^2$. При такой реализации очереди с приоритетами время работы алгоритма Дейкстры составляет $O(n^2)$, при этом в нем доминирует время работы процедуры EXTRACT-MIN.

Реализация с помощью бинарной пирамиды

Бинарная пирамида организует данные как бинарное дерево, хранящееся в массиве. ***Бинарное дерево*** представляет собой разновидность графа, но мы называем его вершины ***узлами***, ребра его неориентированные, а каждый узел имеет 0, 1 или 2 узла ниже, которые являются его ***дочерними узлами***. На левой стороне приведенного рисунка показан пример бинарного дерева с пронумерованными узлами. Узлы без дочерних узлов, такие как узлы с 6 по 10, называются ***листьями***[2].



Бинарная пирамида представляет собой бинарное дерево с тремя дополнительными свойствами. Во-первых, дерево полностью заполняется на всех уровнях, за исключением, возможно, самого нижнего, который заполняется слева до некоторой точки. Во-вторых, каждый узел содержит ключ, на рисунке показанный внутри каждого узла. В-третьих, ключи подчиняются ***свойству пирамиды***: ключ каждого узла не превышает ключи его дочерних узлов. Бинарное дерево на рисунке одновременно является бинарной пирамидой.

Бинарную пирамиду можно хранить в массиве, как показано справа на рисунке. Из-за свойства пирамиды узел с минимальным ключом всегда находится в позиции 1. У узла, находящегося в позиции i, его дочерние узлы находятся в позициях $2i$ и $2i+1$, а узел, находящийся в дереве над ним — ***родительский узел***, — в позиции $\lfloor i/2 \rfloor$. Таким образом, когда бинарная пирамида хранится в массиве, перемещение по ее элементам выполняется очень легко.

Бинарная пирамида обладает еще одной важной характеристикой: если она состоит из n узлов, то ее ***высота*** — количество ребер от корня до самого дальнего листа — составляет всего лишь $\lfloor \lg n \rfloor$. Таким образом, путь от корня до листьев, или из листа в корень, можно пройти за время $O(\lg n)$.

---

[2] *Компьютерные специалисты предпочитают рисовать деревья с корнем вверху, направляя ветви вниз, — в отличие от настоящих деревьев, ветви которого идут вверх от корня, находящегося внизу.*

Поскольку бинарные пирамиды имеют высоту $\lfloor \lg n \rfloor$, три операции очереди с приоритетами выполняются в ней за время $O(\lg n)$ каждая. В случае Insert добавляем новый лист в первую доступную позицию. Затем до тех пор, пока ключ в узле больше, чем ключ в родительском по отношению к нему узле, выполняем обмен содержимым[3] узла с содержимым его родителя и перемещаемся на один уровень вверх к корню. Другими словами, содержимое узла "всплывает" к корню до тех пор, пока не будет выполняться свойство пирамиды. Поскольку путь к корню состоит не более чем из $\lfloor \lg n \rfloor$ ребер, выполняется не более $\lfloor \lg n \rfloor - 1$ обменов, так что время работы операции Insert составляет $O(\lg n)$. Чтобы выполнить операцию Decrease-Key, используется та же идея: уменьшение ключа с последующим всплытием содержимого к корню до тех пор, пока не будет выполнено свойство пирамиды, — вновь за время $O(\lg n)$. Чтобы выполнить операцию Extract-Min, сохраните содержимое корня для возврата вызывающей процедуре. Затем возьмите последний лист (узел с наибольшим номером) и поместите его содержимое в корень. Затем "утопите" содержимое корня, обменивая содержимое узла и дочернего узла с меньшим значением ключа, пока не будет выполнено свойство пирамиды. После этого верните сохраненное значение корня. Поскольку путь от корня до листа не превышает $\lfloor \lg n \rfloor$ ребер, выполняется не более $\lfloor \lg n \rfloor - 1$ обменов, так что время работы операции Extract-Min составляет $O(\lg n)$.

Когда алгоритм Дейкстры использует реализацию очереди с приоритетами с помощью бинарной пирамиды, на вставку вершин он затрачивает время $O(n \lg n)$, на операции Extract-Min — то же время $O(n \lg n)$ и наконец на операции Decrease-Key тратится время $O(m \lg n)$ (на самом деле вставка n вершин выполняется за время $\Theta(n)$, поскольку изначально только у вершины s значение *shortest* равно нулю, а у всех остальных вершин значение *shortest* равно ∞). Если граф ***разреженный***, т.е. число ребер m гораздо меньше, чем n^2, реализация очереди с приоритетами с помощью бинарной пирамиды оказывается более эффективной, чем при использовании простого массива. Графы, моделирующие сети дорог, являются разреженными, так как средний перекресток покидают около четырех дорог, а потому m составляет около $4n$. С другой стороны, когда граф оказывается ***плотным*** т.е. когда m близко к n^2, так что в графе имеется много ребер — время $O(m \lg n)$, которое алгоритм Дейкстры затрачивает на вызовы операций Decrease-Key, может сделать его медленнее, чем при реализации очереди с приоритетами с помощью простого массива.

Вот еще одно применение бинарных пирамид: для выполнения сортировки за время $O(n \lg n)$.

Процедура Heapsort(A,n)

Вход:

- A: массив.
- n: количество сортируемых элементов в массиве A.

Выход: массив B, содержащий элементы массива A в отсортированном порядке.

---

[3] *Содержимое узла включает ключ и любую прочую информацию, связанную с ключом, например какая вершина соответствует этому узлу.*

1. Построить бинарную пирамиду Q из элементов массива A.
2. Пусть $B[1..n]$ представляет собой новый массив.
3. Для $i = 1$ до n:
   A. Вызывать EXTRACT-MIN(Q) и присвоить возвращаемое значение элементу $B[i]$.
4. Вернуть массив B.

Шаг 1 преобразует входной массив в бинарную пирамиду, что можно сделать одним из двух способов. Первый способ заключается в том, чтобы начать с пустой бинарной пирамиды, а затем вставлять каждый элемент массива в сумме за время $O(n \lg n)$. Другой способ состоит в построении бинарной пирамиды непосредственно в массиве, работая снизу вверх, за время $O(n)$. Можно также выполнить сортировку на месте без привлечения дополнительного массива B.

Реализация с помощью фибоначчиевой пирамиды

Можно реализовать очередь с приоритетами и с помощью сложной структуры данных под названием "фибоначчиева пирамида". При использовании фибоначчиевой пирамиды n операций INSERT и EXTRACT-MIN в сумме выполняются за время $O(n \lg n)$, а m операций DECREASE-KEY занимают время $\Theta(m)$, так что общее время работы алгоритма Дейкстры составляет $O(n \lg n + m)$. На практике фибоначчиевы пирамиды используются редко. Во-первых, отдельные операции могут выполняться гораздо дольше, чем в среднем, хотя общее время остается приведенным выше. Во-вторых, фибоначчиевы пирамиды достаточно сложны, так что скрытые в асимптотических обозначениях константные множители существенно бо́льшие, чем в случае бинарных пирамид.

Алгоритм Беллмана–Форда

Если некоторые ребра имеют отрицательные веса, алгоритм Дейкстры может вернуть неверный результат. Алгоритм Беллмана–Форда[4] в состоянии справиться с отрицательными весами ребер, и его можно использовать для обнаружения и помощи в идентификации цикла с отрицательным весом.

Алгоритм Беллмана–Форда удивительно прост. После инициализации значений *shortest* и *pred* он просто ослабляет все m ребер $n-1$ раз. Сама процедура показана ниже, а на рисунке показано, как этот алгоритм работает с небольшим графом. Исходная вершина — s, значения *shortest* приведены в вершинах графа, а заштрихованные ребра указывают значения *pred*: если ребро (u,v) заштриховано, то $pred[v] = u$. В приведенном примере мы предполагаем, что каждый проход по всем ребрам ослабляет их в фиксированном порядке (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y). В части (а) показана ситуация незадолго до первого прохода, а в частях (б)–(д) показано состояние дел после каждого очередного прохода. Значения *shortest* и *pred* в части (д) являются конечными.

---

[4] *Основан на отдельных алгоритмах, разработанных Ричардом Беллманом (Richard Bellman) в 1958 году и Лестером Фордом (Lester Ford) в 1962 году.*

Процедура BELLMAN-FORD(G,s)

Вход:

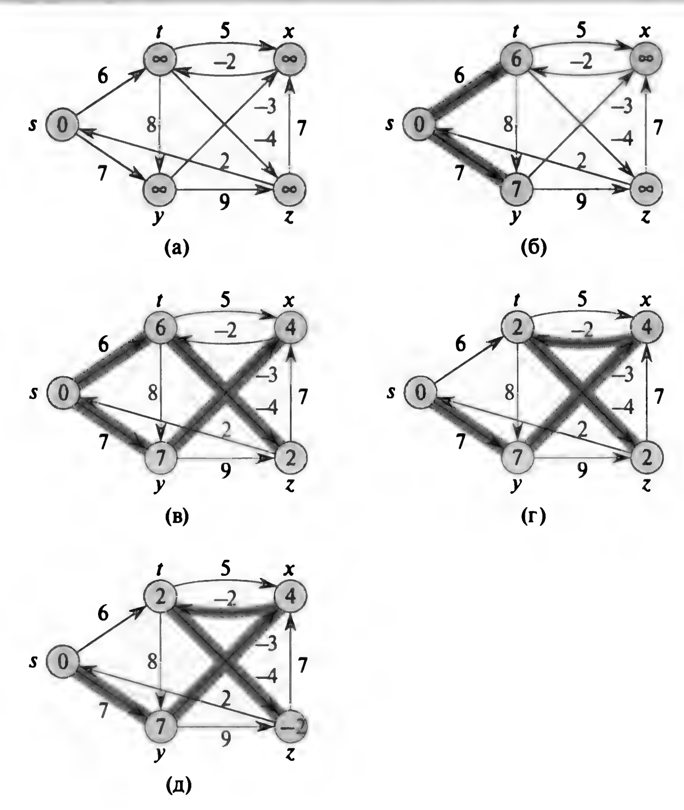
- G: ориентированный граф, содержащий множество V из n вершин и множество E из m ориентированных ребер с произвольными весами.
- s: исходная вершина в V.

Результат: тот же, что и в процедуре DIJKSTRA (с. 100).

1. Установить $shortest[v] = \infty$ для всех вершин v, за исключением s; установить $shortest[s] = 0$ и для всех вершин v установить $pred[v] = \text{NULL}$.
2. Для $i = 1$ до $n-1$:

   A. Для каждого ребра (u,v) из E:

      i. Вызвать RELAX(u,v).



Как может настолько простой алгоритм выдать правильный ответ? Рассмотрим кратчайший путь от источника s к любой вершине v. Вспомним (с. 93), что если мы ослабим ребра по порядку вдоль кратчайшего пути от s к v, то значения $shortest[v]$ и $pred[v]$ окажутся верными. Сейчас, если циклы с отрицательным весом запрещены, всегда имеется кратчайший путь от s к v, не содержащий цикл. Почему? Предположим, что кратчайший путь от s к v содержит цикл. Поскольку этот цикл должен иметь неотрицательный вес,

мы могли бы вырезать этот цикл из пути и в конечном итоге остаться с путем от s к v, вес которого не выше веса пути, содержащего цикл. Каждый ациклический путь должен содержать не более $n-1$ ребер, так как если путь содержит n ребер, то некоторые вершины он должен пройти дважды, что образует цикл. Таким образом, если в графе есть кратчайший путь от s к v, то есть такой кратчайший путь, который содержит не более $n-1$ ребер. При первом выполнении шага 2А ослабляются все ребра, а значит, и первое ребро на кратчайшем пути. При втором выполнении шага 2А вновь ослабляются все ребра, а значит, и второе ребро на кратчайшем пути и т.д. После $(n-1)$-го выполнения все ребра на кратчайшем пути в соответствующем порядке гарантированно ослаблены, а значит, значения $shortest[v]$ и $pred[v]$ оказываются верными.

Теперь предположим, что граф содержит цикл с отрицательным весом, но мы запустили для него процедуру Bellman-Ford. В результате можно ходить и ходить по этому циклу, каждый раз получая все меньший и меньший вес кратчайшего пути. Это означает, что имеется по крайней мере одно ребро (u,v) цикла, для которого $shortest[v]$ будет постоянно уменьшаться при очередном ослаблении — несмотря даже на то, что это ребро уже было ослаблено $n-1$ раз.

Так что вот как можно найти цикл с отрицательным весом, если таковой существует, после запуска процедуры Bellman-Ford. Пройдем по ребрам еще один раз. Если мы найдем ребро (u,v), для которого $shortest[u]+weight(u,v) < shortest[v]$, то мы знаем, что вершина v либо входит в цикл с отрицательным весом, либо достижима из него. Найти вершину в цикле с отрицательным весом можно путем отслеживания значений *pred* на обратном пути от v, отслеживая все посещенные вершины до тех пор, пока не обнаружим ранее посещенную вершину x. Затем мы можем проследить значения *pred*, ведущие обратно из x, до тех пор, пока вновь не достигнем x. Все вершины на этом пути вместе с x образуют цикл с отрицательным весом. Приведенная далее процедура поиска цикла с отрицательным весом показывает, как определить, имеет ли граф такой цикл, и если имеет, то как его построить.

Процедура Find-Negative-Weight-Cycle(G)

Вход: G: ориентированный граф, содержащий множество V из n вершин и множество E из m ориентированных ребер с произвольными весами, для которого уже была выполнена процедура Bellman-Ford.

Выход: либо список вершин, в указанном порядке образующих цикл с отрицательным весом, либо пустой список, если таких циклов в графе нет.

1. Пройти по всем ребрам в поисках такого ребра (u,v), что $shortest[u]+weight(u,v) < shortest[v]$.
2. Если таких ребер нет, вернуть пустой список.
3. В противном случае (имеется некоторое ребро (u,v), такое, что $shortest[u]+weight(u,v) < shortest[v]$) выполнить следующее.
   А. Пусть *visited* — новый массив с одним элементом для каждой вершины. Установить все элементы массива *visited* равными FALSE.

B. Установить x равным v.

C. Пока $visited[x]$ равно FALSE, выполнять следующее.

i. Установить $visited[x] =$ TRUE.

ii. Установить x равным $pred[x]$.

D. Теперь мы знаем, что x представляет собой вершину в цикле с отрицательным весом. Установить v равным $pred[x]$.

E. Создать список вершин *cycle*, изначально содержащий только вершину x.

F. Пока v не равно x, выполнять следующее.

i. Вставить вершину v в начало цикла.

ii. Установить v равным $pred[v]$.

G. Вернуть список *cycle*.

Проанализировать время работы алгоритма Беллмана–Форда достаточно легко. Цикл на шаге 2 выполняет свои итерации $n-1$ раз, и каждый раз при таком выполнении цикл на шаге 2A выполняет m итераций, по одной для каждого ребра. Общее время работы, таким образом, оказывается равным $\Theta(nm)$. Чтобы выяснить, существует ли цикл с отрицательным весом, каждое ребро ослабляется еще раз либо до тех пор, пока ослабление не изменит значение *shortest*, либо пока не будут ослаблены все ребра, что выполняется за время $O(m)$. При наличии цикла с отрицательным весом он не может состоять более чем из n ребер, так что время его трассировки составляет $O(n)$.

В начале этой главы говорилось о том, что циклы с отрицательным весом могут возникать в области обмена валют. Обменные курсы валют меняются быстро. Представьте себе, что в некоторый момент времени действуют следующие курсы валют.

За 1 доллар США можно купить 0.7292 евро.

За 1 евро можно купить 105.374 японской иены.

За 1 японскую иену можно купить 0.3931 российского рубля.

За 1 российский рубль можно купить 0.0341 доллара США.

Тогда вы могли бы взять 1 доллар США, купить за него 0.7292 евро, взять 0.7292 евро и купить 76.8387 иены (поскольку $0.7292 \cdot 105.374 = 76.8387$ с точностью до четырех знаков после запятой), взять 76.8387 иены и купить 30.2053 рубля (поскольку $76.8387 \cdot 0.3931 = 30.2053$ с точностью до четырех знаков после запятой) и наконец взять 30.2053 рубля и купить 1.03 доллара (поскольку $30.2053 \cdot 0.0341 = 1.0300$ с точностью до четырех знаков после запятой). Если все четыре операции можно выполнить до изменения обменных курсов, то можно получить 3% прибыли. Начните с одного миллиона долларов, и вы получите прибыль в 30 тысяч долларов, не пошевелив пальцем!

Такой сценарий называется возможностью арбитража. Вот как найти ее путем поиска цикла с отрицательным весом. Предположим, что имеются n валют $c_1, c_2, c_3, \ldots, c_n$ и известны все обменные курсы между парами валют. Предположим, что за единицу валюты c_i можно купить r_{ij} единиц валюты c_j, так что r_{ij} представляет собой обменный курс валют c_i и c_j. И i, и j находятся в диапазоне от 1 до n (считаем, что $r_{ii} = 1$ для всех валют c_{ij}).

Возможность арбитража соответствовать такой последовательности k валют $\langle c_{j_1}, c_{j_2}, c_{j_3}, \ldots, c_{j_k} \rangle$, что когда вы перемножаете их обменные курсы, то получаете произведение, строго большее 1:

$$r_{j_1,j_2} \cdot r_{j_2,j_3} \cdots r_{j_{k-1},j_k} \cdot r_{j_k,j_1} > 1.$$

Прологарифмируем обе части неравенства. Основание логарифмов значения не имеет, так что поступим, как настоящие компьютерщики, и будем логарифмировать по основанию 2. Так как логарифм произведения равен сумме логарифмов сомножителей, т.е. $\lg(x \cdot y) = \lg x + \lg y$, нас интересует ситуация, когда

$$\lg r_{j_1,j_2} + \lg r_{j_2,j_3} + \cdots + \lg r_{j_{k-1},j_k} + \lg r_{j_k,j_1} > 0.$$

Меняя знаки с обеих сторон неравенства, получаем

$$(-\lg r_{j_1,j_2}) + (-\lg r_{j_2,j_3}) + \cdots + (-\lg r_{j_{k-1},j_k}) + (-\lg r_{j_k,j_1}) < 0,$$

что соответствует циклу с весами ребер, равными взятым с обратными знаками логарифмам обменных курсов.

Чтобы найти возможность арбитража, если таковая существует, надо построить ориентированный граф с вершиной v_i для каждой валюты c_i. Для каждой пары валют c_i и c_j создаются ориентированные ребра (v_i, v_j) и (v_j, v_i) с весами $-\lg r_{ij}$ и $-\lg r_{ji}$ соответственно. Добавим новую вершину s с ребрами (s, v_i) с нулевым весом, идущими к каждой вершине от v_1 до v_n. Затем выполним алгоритм Беллмана–Форда над этим графом с s в качестве исходной вершины, а результатом воспользуемся для выяснения, не содержит ли он цикл с отрицательным весом. Если содержит, то вершины этого цикла соответствуют валютам возможности арбитража. Общее количество ребер m равно $n + n(n-1) = n^2$, так что алгоритм Беллмана–Форда выполняется за время $O(n^3)$; кроме того, время $O(n^2)$ требуется для выяснения наличия цикла с отрицательным весом и $O(n)$ — для его идентификации (если он существует). Хотя время $O(n^3)$ кажется слишком большим, на практике все не так плохо, потому что константные множители, скрывающиеся в асимптотических обозначениях, достаточно малы. Я написал и скомпилировал соответствующую программу и выполнил ее на моем MacBook Pro 2,4 ГГц для 182 валют — по количеству валют всего мира. После загрузки обменных курсов (я воспользовался генератором случайных чисел) программа завершилась примерно за 0.02 секунды.

Алгоритм Флойда–Уоршелла

Теперь предположим, что вы хотите найти кратчайший путь от каждой вершины графа к каждой другой вершине. Это задача поиска ***кратчайших путей между всеми парами вершин***.

Классический пример кратчайших путей между всеми парами вершин, на который ссылается множество авторов, — это таблица в дорожном атласе, указывающая расстояния между городами. Найдите строку для одного города, столбец для другого, и на пересечении этих строки и столбца вы найдете расстояние между указанными городами.

У этого примера имеется только одна небольшая проблема — это *не все пары*. Если бы это были кратчайшие расстояния между всеми парами, таблица содержала бы по одной строке и одному столбцу для каждого перекрестка, а не только для каждого города. Число

строк и столбцов в этом случае оказалось бы равным миллионам, так что единственный способ сделать эту таблицу удобочитаемой — сократить ее и указывать кратчайшие пути только между городами.

Но вот пример применения поиска кратчайших путей между всеми парами вершин без сучка и задоринки: поиск ***диаметра*** сети, который представляет собой самый длинный из всех кратчайших путей. Например, предположим, что ориентированный граф представляет собой коммуникационную сеть, а вес ребра определяет время, необходимое сообщению для прохода по каналу связи. Тогда диаметр дает наибольшее время прохода сообщения через сеть.

Конечно, можно вычислить кратчайшие пути между всеми парами вершин путем вычисления кратчайших путей от одного источника поочередно для каждой вершины. Если все ребра имеют неотрицательные веса, можно воспользоваться алгоритмом Дейкстры для каждой из n вершин, и каждый такой вызов выполняется за время $O(m \lg n)$ при использовании бинарных пирамид, и $O(n \lg n + m)$ при использовании фибоначчиевых пирамид. Так что общее время работы программы будет составлять либо $O(nm \lg n)$, либо $O(n^2 \lg n + nm)$. Если граф разреженный, такой подход вполне работоспособен. Но если граф плотный, так что m близко к n^2, то $O(nm \lg n)$ превращается в $O(n^3 \lg n)$. Даже применение фибоначчиевых пирамид дает для плотного графа время работы $O(n^3)$, и при этом скрытый постоянный множитель в асимптотической записи может быть значительным из-за сложности реализации фибоначчиевых пирамид. Конечно, если граф может содержать ребра с отрицательным весом, то говорить о применении алгоритма Дейкстры не приходится, и для каждой из n вершин следует использовать алгоритм Беллмана–Форда, время работы которого $\Theta(n^2 m)$ для плотного графа превращается в $\Theta(n^4)$.

Если же вместо этого воспользоваться алгоритмом Флойда–Уоршелла[5], то задачу поиска кратчайших путей между всеми парами вершин можно решить за время $\Theta(n^3)$, независимо от того, плотный ли граф, разреженный или представляет собой некоторый средний случай. Алгоритм допускает наличие в графе ребер (но не циклов) с отрицательным весом. При этом скрытая в Θ-обозначении константа достаточно мала. Кроме прочего, алгоритм Флойда–Уоршелла демонстрирует применение алгоритмического метода, именуемого "динамическим программированием".

Алгоритм Флойда–Уоршелла основан на очевидном свойстве кратчайших путей. Предположим, что вы ведете машину от Киева до Запорожья по кратчайшему маршруту и что этот кратчайший маршрут проходит через Переяслав-Хмельницкий и Кременчуг. Тогда часть кратчайшего пути из Киева в Запорожье, идущая из Переяслава-Хмельницкого в Кременчуг, должна быть кратчайшим путем из Переяслава-Хмельницкого в Кременчуг. Почему? Потому что, если бы был более короткий маршрут из Переяслава-Хмельницкого в Кременчуг, мы использовали бы его в кратчайшем пути из Киева в Запорожье! Как я и говорил, это совершенно очевидное свойство. Применим этот принцип к ориентированным графам.

---

[5] *Назван по именам его разработчиков, Роберта Флойда (Robert Floyd) и Стивена Уоршелла (Stephen Warshall).*

Если кратчайший путь (назовем его p) из вершины u в вершину v идет из вершины u через вершину x, затем в вершину y, а потом в вершину v, то часть пути p между вершинами x и y сама является кратчайшим путем из x в y. То есть *любой подпуть кратчайшего пути является кратчайшим путем*.

Алгоритм Флойда–Уоршелла отслеживает веса путей и предшественников вершин в массивах, индексируемых в трех измерениях. Одномерный массив можно представить в виде таблицы (такой, как, например, на с. 24). Двумерный массив можно представить в виде матрицы (как, например, матрица смежности на с. 86). Для идентификации записи в этом случае необходимы два индекса — строка и столбец. Двумерный массив можно представить также как одномерный массив, в котором каждая запись сама по себе является одномерным массивом. Трехмерный массив можно рассматривать как одномерный массив двумерных массивов; для идентификации записи необходимы значения индексов в каждом из трех измерений. Для разделения размерностей при индексации многомерных массивов мы будем использовать запятые.

В алгоритме Флойда–Уоршелла мы предполагаем, что вершины пронумерованы от 1 до n. Номера вершин в данном случае важны, потому что алгоритм Флойда–Уоршелла использует следующее определение.

$shortest[u,v,x]$ представляет собой вес кратчайшего пути от вершины u к вершине v, в котором все промежуточные вершины — вершины на пути, отличные от u и v — пронумерованы от 1 до x.

(Так что u, v и x можно рассматривать как целые числа в диапазоне от 1 до n, которые представляют вершины.) Это определение не требует, чтобы промежуточные вершины включали *все* x вершин с номерами от 1 до x; оно просто требует, чтобы каждая промежуточная вершина — сколько их ни есть — имела номер не выше x. Поскольку все вершины имеют номер не выше n, значение $shortest[u,v,n]$ должно быть равно $sp(u,v)$, весу кратчайшего пути от u к v.

Рассмотрим две вершины, u и v, и выберем число x в диапазоне от 1 до n. Рассмотрим также все пути от u к v, в которых все промежуточные вершины имеют номера, не превышающие x. Пусть среди всех этих путей путь p имеет наименьший вес. Путь p либо включает вершину x, либо нет, и мы знаем, что, за исключением, возможно, u или v, он не содержит ни одну вершину с номером больше x. Существуют две возможности.

- Первая возможность: x не является промежуточной вершиной пути p. Тогда все промежуточные вершины пути p имеют номера не более $x-1$. Что это значит? Это значит, что вес кратчайшего пути от u к v со всеми промежуточными вершинами с номерами не более x, такой же, как и вес кратчайшего пути от u к v, в котором все промежуточные вершины имеют номера не выше $x-1$. Иными словами, $shortest[u,v,x]$ равно $shortest[u,v,x-1]$.

- Вторая возможность: x появляется в качестве промежуточной вершины пути p. Поскольку любой подпуть кратчайшего пути сам по себе является кратчайшим путем,

часть пути p, которая идет от u к x, является кратчайшим путем от u к x. Аналогично часть p, которая идет от x к v, является кратчайшим путем от x к v. Поскольку вершина x является конечной точкой каждого из этих подпутей, она не может быть промежуточной вершиной в любом из них, а значит, промежуточные вершины в каждом из рассмотренных подпутей имеют номера, не превышающие $x-1$. Таким образом, вес кратчайшего пути от u к v, в котором все промежуточные вершины имеют номера, не превышающие x, равен сумме весов двух кратчайших путей: одного — от u к x со всеми промежуточными вершинами с номерами не выше $x-1$ и второго — от x к v, также со всеми промежуточными вершинами с номерами не выше $x-1$. Иными словами, $shortest[u,v,x]$ равно сумме $shortest[u,x,x-1]+shortest[x,v,x-1]$.

Поскольку x либо является промежуточной вершиной на кратчайшем пути от u к v, либо нет, мы можем заключить, что $shortest[u,v,x]$ является меньшим из значений $shortest[u,x,x-1]+shortest[x,v,x-1]$ и $shortest[u,v,x-1]$.

Наилучший способ представления графа в алгоритме Флойда–Уоршелла — с помощью варианта представления матрицей смежности. В такой матрице каждый элемент уже не ограничен значениями 0 или 1; запись для ребра (u,v) содержит вес ребра, где значение ∞ означает, что данное ребро отсутствует. Поскольку $shortest[u,v,0]$ означает вес кратчайшего пути с промежуточными вершинами с номерами не более 0, такой путь не имеет промежуточных вершин, т.е. состоит только из одного ребра. Таким образом, описанная нами матрица — именно то, что требуется для $shortest[u,v,0]$.

Исходя из имеющихся значений $shortest[u,v,0]$ (которые представляют собой веса ребер), алгоритм Флойда–Уоршелла вычисляет значения $shortest[u,v,x]$ сначала для всех пар вершин u и v с x, равным 1. После этого алгоритм вычисляет значения $shortest[u,v,x]$ сначала для всех пар вершин u и v с x, равным 2, а затем — с x, равным 3 и так далее вплоть до n.

Как отслеживать предшественников вершин на кратчайших путях? Определим $pred[u,v,x]$ аналогично тому, как мы определили $shortest[u,v,x]$ как предшественника вершины v на кратчайшем пути от вершины u, в котором номера всех промежуточных вершин не превышают x. Мы можем обновлять значения $pred[u,v,x]$ так же, как мы вычисляли значения $shortest[u,v,x]$. Если $shortest[u,v,x]$ имеет то же значение, что и $shortest[u,v,x-1]$, то кратчайший путь от u к v, в котором номера всех промежуточных вершин не превышают x, такой же, как и путь, в котором номера всех промежуточных вершин не превышают $x-1$. Предшественники вершины v при этом должны быть одинаковы в обоих путях, так что $pred[u,v,x]$ мы делаем таким же, как и $pred[u,v,x-1]$. Но что если $shortest[u,v,x]$ меньше $shortest[u,v,x-1]$? Это происходит, когда мы находим путь от u к v, имеющий в качестве промежуточной вершину x и имеющий меньший вес, чем кратчайший путь от u к v, в котором номера всех промежуточных вершин не превышают $x-1$. Поскольку x должна быть промежуточной вершиной на этом вновь найденном кратчайшем пути, предшественник вершины v на пути от u должен быть тем же, что и предшественник v на пути от x. В этом случае мы устанавливаем $pred[u,v,x]$ равным $pred[x,v,x-1]$.

Теперь мы можем сложить все кусочки мозаики в единую картину алгоритма Флойда–Уоршелла.

Процедура FLOYD-WARSHALL(G)

Вход: G: граф, представленный весовой матрицей смежности W с n строками и n столбцами (по одной строке и одному столбцу для каждой вершины). Запись в строке u и столбце v, записываемая как w_{uv}, представляет собой вес ребра (u,v), если таковое присутствует в G, и равна ∞ в противном случае.

Выход: для каждой пары вершин u и v значение $shortest[u,v,n]$ содержит вес кратчайшего пути от u до v, а $pred[u,v,n]$ — вершину, являющуюся предшественником вершины v на кратчайшем пути из u.

1. Пусть *shortest* и *pred* представляют собой новые массивы размером $n \times n \times (n+1)$.
2. Для всех u и v от 1 до n:

   A. Установить $shortest[u,v,0]$ равным w_{uv}.

   B. Если (u,v) является ребром графа G, установить $pred[u,v,0]$ равным u. В противном случае установить $pred[u,v,0]$ равным NULL.
3. Для $x = 1$ до n:

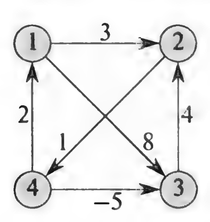
   A. Для $u = 1$ до n:

      i. Для $v = 1$ до n:

         a. Если $shortest[u,v,x] < shortest[u,x,x-1] + shortest[x,v,x-1]$, то установить $shortest[u,v,x]$ равным $shortest[u,x,x-1] + shortest[x,v,x-1]$, а $pred[u,v,x]$ равным $pred[x,v,x-1]$.

         b. В противном случае установить $shortest[u,v,x]$ равным $shortest[u,v,x-1]$, а $pred[u,v,x]$ равным $pred[u,v,x-1]$.
4. Вернуть массивы *shortest* и *pred*.

Для графа



матрица смежности W, содержащая веса ребер, имеет вид

$$\begin{pmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & \infty & 1 \\ \infty & 4 & 0 & \infty \\ 2 & \infty & -5 & 0 \end{pmatrix}$$

и дает значения $shortest[u,v,0]$[6] (веса путей не более чем с одним ребром). Например, $shortest[2,4,0]$ равно 1, поскольку из вершины 2 можно попасть в вершину 4 непосредственно, без промежуточных вершин, по ребру $(2,4)$, имеющему вес 1. Аналогично $shortest[4,3,0]$ равно −5. А вот как выглядит матрица $pred[u,v,0]$:

$$\begin{pmatrix} \text{NULL} & 1 & 1 & \text{NULL} \\ \text{NULL} & \text{NULL} & \text{NULL} & 2 \\ \text{NULL} & 3 & \text{NULL} & \text{NULL} \\ 4 & \text{NULL} & 4 & \text{NULL} \end{pmatrix}.$$

Например, $pred[2,4,0]$ равно 2, поскольку предшественником вершины 4 является вершина 2, при использовании ребра $(2,4)$, имеющего вес 1. $pred[2,3,0]$ равно NULL, поскольку ребра $(2,3)$ в графе нет.

После выполнения цикла на шаге 3 для $x = 1$ (рассматриваются пути, которые могут включать вершину 1 в качестве промежуточной) значения $shortest[u,v,1]$ и $pred[u,v,1]$ имеют вид

$$\begin{pmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & \infty & 1 \\ \infty & 4 & 0 & \infty \\ 2 & 5 & -5 & 0 \end{pmatrix} \text{ и } \begin{pmatrix} \text{NULL} & 1 & 1 & \text{NULL} \\ \text{NULL} & \text{NULL} & \text{NULL} & 2 \\ \text{NULL} & 3 & \text{NULL} & \text{NULL} \\ 4 & 1 & 4 & \text{NULL} \end{pmatrix}.$$

После выполнения цикла для $x = 2$ значения $shortest[u,v,2]$ и $pred[u,v,2]$ имеют вид

$$\begin{pmatrix} 0 & 3 & 8 & 4 \\ \infty & 0 & \infty & 1 \\ \infty & 4 & 0 & 5 \\ 2 & 5 & -5 & 0 \end{pmatrix} \text{ и } \begin{pmatrix} \text{NULL} & 1 & 1 & 2 \\ \text{NULL} & \text{NULL} & \text{NULL} & 2 \\ \text{NULL} & 3 & \text{NULL} & 2 \\ 4 & 1 & 4 & \text{NULL} \end{pmatrix}.$$

После выполнения цикла для $x = 3$ значения $shortest[u,v,3]$ и $pred[u,v,3]$ имеют вид

$$\begin{pmatrix} 0 & 3 & 8 & 4 \\ \infty & 0 & \infty & 1 \\ \infty & 4 & 0 & 5 \\ 2 & -1 & -5 & 0 \end{pmatrix} \text{ и } \begin{pmatrix} \text{NULL} & 1 & 1 & 2 \\ \text{NULL} & \text{NULL} & \text{NULL} & 2 \\ \text{NULL} & 3 & \text{NULL} & 2 \\ 4 & 3 & 4 & \text{NULL} \end{pmatrix}.$$

Окончательные значения $shortest[u,v,4]$ и $pred[u,v,4]$ после выполнения цикла для $x = 4$ имеют вид

$$\begin{pmatrix} 0 & 3 & -1 & 4 \\ 3 & 0 & -4 & 1 \\ 7 & 4 & 0 & 5 \\ 2 & -1 & -5 & 0 \end{pmatrix} \text{ и } \begin{pmatrix} \text{NULL} & 1 & 4 & 2 \\ 4 & \text{NULL} & 4 & 2 \\ 4 & 3 & \text{NULL} & 2 \\ 4 & 3 & 4 & \text{NULL} \end{pmatrix}.$$

[6] *Поскольку трехмерный массив представляет собой одномерный массив двумерных массивов, для фиксированного значения x мы рассматриваем* $\mathtt{shortest}[u,v,x]$ *как двумерный массив.*

Отсюда видно, например, что кратчайший путь из вершины 1 в вершину 3 имеет вес -1. Этот путь начинается в вершине 1, проходит через вершины 2 и 4 и завершается в вершине 3, что можно отследить по значениям массива *pred*: $pred[1,3,4]=4$, $pred[1,4,4]=2$ и $pred[1,2,4]=1$.

Я говорил, что время работы алгоритма Флойда–Уоршелла равно $\Theta(n^3)$, и в этом легко убедиться. У нас есть три вложенных один в другой цикла, и каждый из них выполняет по n итераций. В каждой итерации цикла на шаге 3 цикл на шаге 3A выполняет n итераций. Аналогично в каждой итерации цикла на шаге 3A цикл на шаге 3Ai выполняет n итераций. Поскольку внешний цикл на шаге 3 также выполняет n итераций, внутренний цикл (на шаге 3Ai) выполняет всего n^3 итераций. Каждая итерация самого глубоко вложенного цикла выполняется за константное время, так что общее время работы алгоритма равно $\Theta(n^3)$.

Что касается используемой памяти, то дело выглядит так, будто этот алгоритм требует $\Theta(n^3)$ памяти. В конце концов, он ведь создает два массива размером $n \times n \times (n+1)$. А поскольку каждая запись массива занимает константное количество памяти, в сумме эти массивы занимают $\Theta(n^3)$ памяти. Оказывается, однако, что можно ограничиться памятью всего лишь размером $\Theta(n^2)$. Каким образом? Просто создав массивы *shortest* и *pred* размером $n \times n$ и забыв о третьем индексе. Хотя на шагах 3Aia и 3Aib происходит обновление одних и тех же значений $shortest[u,v]$ и $pred[u,v]$, оказывается, что в конце эти массивы содержат верные значения!

Ранее я упоминал, что алгоритм Флойда–Уоршелла иллюстрирует применение технологии ***динамического программирования***. Эта технология применима, когда

1. мы пытаемся найти оптимальное решение задачи,
2. мы можем разбить экземпляр задачи на экземпляры одной или нескольких подзадач,
3. мы используем решения подзадач (или подзадачи) для решения исходной задачи,
4. если мы используем решение подзадачи в оптимальном решении исходной задачи, то решение используемой подзадачи должно быть оптимальным.

Мы можем подытожить эти условия единым лаконичным названием ***оптимальная подструктура***. Говоря кратко, оптимальное решение проблемы должно содержать в себе оптимальные решения подзадач. В динамическом программировании у нас есть некоторое понятие "размера" подзадачи, и мы часто решаем подзадачи в порядке увеличения размера, т.е. сначала решаем маленькие подзадачи, затем, как только у нас имеются оптимальные решения меньших подзадач, можно пытаться решать большие подзадачи, используя оптимальные решения меньших подзадач оптимальным же образом.

Такое описание динамического программирования выглядит довольно абстрактно, так что давайте посмотрим, как его использует алгоритм Флойда–Уоршелла. Сформулируем подзадачу как

вычислить значение $shortest[u,v,x]$, которое представляет собой вес кратчайшего пути от вершины u до вершины v, в котором промежуточные вершины имеют номера от 1 до x.

Здесь "размер" подзадачи — наибольший номер вершины, которая может быть промежуточной на кратчайшем пути; иными словами, это значение x. Оптимальная подструктура осуществляется благодаря следующему свойству.

Рассмотрим кратчайший путь p от вершины u к вершине v, и пусть x — наибольший номер промежуточной вершины на этом пути. Тогда часть p, идущая от u к x, представляет собой кратчайший путь от u к x, в котором все промежуточные вершины имеют номера, не превышающие $x-1$, а часть p, идущая от x к v, представляет собой кратчайший путь от x к v, в котором все промежуточные вершины имеют номера, не превышающие $x-1$.

Мы решаем задачу вычисления $shortest[u,v,x]$, сначала вычисляя $shortest[u,v,x-1]$, $shortest[u,x,x-1]$ и $shortest[x,v,x-1]$, а затем используя меньшее из значений $shortest[u,v,x-1]$ и $shortest[u,x,x-1]+shortest[x,v,x-1]$. Поскольку мы должны вычислить все значения, где третий индекс равен $x-1$, прежде чем вычислять значения с третьим индексом x, у нас имеется вся информация, необходимая для вычисления $shortest[u,v,x]$.

Обычной практикой динамического программирования является хранение оптимальных решений подзадач ($shortest[u,v,x-1]$, $shortest[u,x,x-1]$ и $shortest[x,v,x-1]$) в таблице, а затем просмотр таблицы при вычислении оптимального решения исходной задачи ($shortest[u,v,x]$). Такой подход называется восходящим, так как работает от меньших подзадач к бо́льшим. Другой подход заключается в нисходящем решении подзадач, переходя от бо́льших подзадач к более мелким (сохраняя при этом результат решения каждой подзадачи в таблице).

Динамическое программирование применимо к широкому спектру задач оптимизации, и лишь некоторые из них связаны с графами. Мы вновь встретимся с ним в главе 7, "Алгоритмы на строках", когда будем искать наибольшую общую подпоследовательность двух символьных строк.

Дальнейшее чтение

Алгоритмы Дейкстры и Беллмана–Форда подробно описаны в главе 24 CLRS [4]. В главе 25 CLRS рассматриваются алгоритмы поиска кратчайших путей между всеми парами вершин, включая алгоритм Флойда–Уоршелла; все алгоритмы поиска кратчайших путей между всеми парами вершин, основанные на умножении матриц, имеют время работы $\Theta(n^3 \lg n)$; интеллектуальный алгоритм Дональда Джонсона (Donald Johnson), разработанный для поиска кратчайших путей между всеми парами вершин в разреженных графах, имеет время работы $O(n^2 \lg n + nm)$.

Когда веса ребер представляют собой небольшие неотрицательные целые числа, не превышающие некоторую известную величину C, более сложная реализация очереди с приоритетами в алгоритме Дейкстры дает асимптотическое время работы, превосходящее использование фибоначчиевых пирамид. Например, Джа (Ahuja), Мельхорн (Mehlhorn), Орлин (Orlin) и Таржан (Tarjan) [2] включили в алгоритм Дейкстры "перераспределенные пирамиды", что дало время работы алгоритма, равное $O(m + n\sqrt{\lg C})$.

7... Алгоритмы на строках

Строка представляет собой просто последовательность символов из некоторого их набора. Например, эта книга включает в себя символы из множества букв, цифр, знаков препинания и математических символов, которое является довольно большим, но конечным. Биологи кодируют ДНК как строки всего лишь из четырех символов — A, C, G, T, — которые представляют базовые аминокислоты аденин, цитозин, гуанин и тимин.

Мы можем выяснять различную информацию о строках, но в этой главе мы остановимся на алгоритмах для решения трех задач, для которых строки являются входными данными:

1. найти наибольшую общую подпоследовательность символов двух строк;
2. для заданного множества операций, преобразующих строку, и стоимости каждой операции найти способ преобразования одной строки в другую с наименьшей стоимостью;
3. найти все вхождения заданной строки в другую строку или текст.

Первые две из этих задач находят применение в вычислительной биологии. Чем длиннее общая подпоследовательность аминокислот, которую мы можем найти у двух ДНК, тем больше они похожи. Другой способ сравнения ДНК состоит в преобразовании одной в другую; чем ниже стоимость такого преобразования, тем более они сходны. Последняя задача — поиск вхождения строки в текст (которая также часто называется ***сопоставлением строк***) — используется в программах всех видов, например где в меню есть команда "Поиск". Она также находит применение в вычислительной биологии, где мы можем искать одну цепочку ДНК внутри другой.

Наидлиннейшая общая подпоследовательность

Начнем с того, что мы подразумеваем под терминами "последовательность" и "подпоследовательность". ***Последовательность*** представляет собой список элементов, в которых играет роль их порядок. Определенный элемент может появляться в последовательности несколько раз. Последовательности, с которыми мы будем работать в этой главе, представляют собой строки символов, и мы будем использовать вместо термина "последовательность" термин "строка". Аналогично мы предполагаем, что элементы, составляющие последовательность, являются символами. Например, строка GACA содержит один и тот же символ (A) несколько раз и отличается от строки CAAG, которая состоит из тех же символов, но в другом порядке.

Подпоследовательностью Z строки X является строка X, возможно, с удаленными элементами. Например, если X является строкой GAC, то он имеет восемь подпоследовательностей: GAC (без удаленных символов), GA (удален C), GC (удален A), AC (удален G), G (удалены A и C), A (удалены G и C), C (удалены G и A) и пустую строку (удалены все символы). Если X и Y являются строками, то Z является ***общей подпоследовательностью***

X и Y, если она является подпоследовательностью обеих строк. Например, если X — это строка CATCGA, а Y является строкой GTACCGTCA, то CCA является общей подпоследовательностью X и Y, состоящей из трех символов. Однако это не наидлиннейшая общая подпоследовательность, поскольку общая подпоследовательность CTCA состоит из четырех символов. Подпоследовательность CTCA действительно является наидлиннейшей общей подпоследовательностью, но не единственной, так как TCGA является еще одной общей подпоследовательностью из четырех символов. Следует различать понятия подпоследовательности и подстроки: ***подстрока*** представляет собой подпоследовательность строки, в которой все символы выбираются из смежных позиций в строке. Для строки CATCGA подпоследовательность ATCG является подстрокой, в то время как подпоследовательность CTCA таковой не является.

Наша цель заключается в том, чтобы для двух заданных строк X и Y найти наидлиннейшую общую подпоследовательность Z строк X и Y. Для этого мы будем использовать динамическое программирование, с которым мы уже встречались в главе 6, "Кратчайшие пути".

Можно найти наидлиннейшую общую подпоследовательность, не прибегая к динамическому программированию, но я не рекомендую поступать таким образом. Можно было бы испытать все подпоследовательности X и проверить, является ли каждая из них подпоследовательностью Y, начиная с самых длинных подпоследовательностей X. Так вы в какой-то момент найдете искомую подпоследовательность (в конечном итоге она всегда имеется, так как пустая строка является подпоследовательностью всех строк). Беда в том, что если длина X равна m, то она имеет 2^m подпоследовательностей, и поэтому даже если игнорировать время проверки каждой из подпоследовательностей, входит ли она в Y, время поиска наидлиннейшей общей подпоследовательности в наихудшем случае будет по меньшей мере экспоненциально зависеть от длины X.

Вспомним из главы 6, "Кратчайшие пути", что для применения динамического программирования требуется оптимальная подструктура: оптимальное решение задачи должно состоять из оптимальных решений ее подзадач. Чтобы найти наидлиннейшую общую подпоследовательность двух строк с помощью динамического программирования, необходимо сначала выяснить, что представляет собой подзадача. Для этого можно воспользоваться префиксами. Если X является строкой $x_1x_2x_3 \cdots x_m$, то ***i-м префиксом*** X является строка $x_1x_2x_3 \cdots x_i$, которую мы будем обозначать как X_i. Величина i должна быть в диапазоне от 0 до m; X_0 является пустой строкой. Например, если строка X — CATCGA, то X_4 — CATC.

Можно увидеть, что наидлиннейшая общая подпоследовательность двух строк содержит наидлиннейшие общие подпоследовательности префиксов этих двух строк. Рассмотрим две строки: $X = x_1x_2x_3 \cdots x_m$ и $Y = y_1y_2y_3 \cdots y_n$. Они имеют некоторую наидлиннейшую общую подпоследовательность, скажем, Z, где $Z = z_1z_2z_3 \cdots z_k$ для некоторой длины k, которая может иметь значение от 0 до меньшего из значений m и n. Что мы можем сказать о Z? Давайте посмотрим на последние символы строк X и Y: x_m и y_n. Они могут быть одинаковыми или не совпадать.

- Если они совпадают, последний символ z_k строки Z должен быть таким же, как и этот символ. Что мы знаем об остальной части строки Z, которой является $Z_{k-1} = z_1 z_2 z_3 \cdots z_{k-1}$? Мы знаем, что Z_{k-1} должна быть наидлиннейшей общей подпоследовательностью того, что осталось от X и Y, а именно — $X_{m-1} = x_1 x_2 x_3 \cdots x_{m-1}$ и $Y_{n-1} = y_1 y_2 y_3 \cdots y_{n-1}$. В нашем более раннем примере, где X — это строка CATCGA, а Y является строкой GTACCGTCA, наидлиннейшая общая подпоследовательность Z = CTCA, и мы видим, что Z_3 = CTC должна быть наидлиннейшей общей подпоследовательностью X_5 = CATCG и Y_8 = GTACCGTC.
- Если они различны, то z_k может быть таким же, как последний символ x_m строки X или последний символ y_n строки Y, но не оба. Кроме того, z_k может не совпадать ни с последним символом X, ни с последним символом Y. Если z_k не совпадает с x_m, игнорируем последний символ X: Z должна быть наидлиннейшей общей подпоследовательностью X_{m-1} и Y. Аналогично, если z_k не совпадает с y_n, игнорируем последний символ Y: Z должна быть наидлиннейшей общей подпоследовательностью X и Y_{n-1}. Продолжая пример, рассмотрим X = CATCG, Y = GTACCGTC и Z = CTC. Здесь z_3 совпадает с y_8 (C), но не с x_5 (G), а потому Z является наидлиннейшей общей подпоследовательностью X_4 = CATC и Y.

Следовательно, рассматриваемая задача имеет оптимальную подструктуру: наидлиннейшая общая подпоследовательность двух строк содержит в себе наидлиннейшие общие подпоследовательности префиксов этих двух строк.

Как же мы должны поступить? Нам необходимо решить одну или две подзадачи, в зависимости от того, совпадают ли последние символы X и Y. Если совпадают, то мы решаем только одну подзадачу — поиска наидлиннейшей общей подпоследовательности X_{m-1} и Y_{n-1}, — а затем добавим к ней этот последний символ, чтобы получить наидлиннейшую общую подпоследовательность X и Y. Если последние символы X и Y не совпадают, то нам надо решить две подзадачи — найти наидлиннейшие общие подпоследовательности X_{m-1} и Y, а также X и Y_{n-1} — и использовать бо́льшую из них в качестве наидлиннейшей общей подпоследовательности X и Y. Если их длины одинаковы, можно использовать любую из них — конкретный выбор не имеет значения.

Мы будем решать задачу поиска наидлиннейшей общей подпоследовательности X и Y в два этапа. Во-первых, мы найдем длину наидлиннейшей общей подпоследовательности X и Y, а также длины соответствующих подпоследовательностей для всех префиксов X и Y. Вас может удивить, что мы ищем длины подпоследовательностей, не зная их самих. Но после вычисления длин мы "реконструируем" эти вычисления, чтобы найти фактическую наидлиннейшую общую подпоследовательность X и Y.

Чтобы быть несколько более точными, давайте обозначим длину наидлиннейшей общей подпоследовательности префиксов X_i и Y_j как $l[i,j]$. Соответственно, длина наидлиннейшей общей подпоследовательности X и Y равна $l[m,n]$. Индексы i и j начинаются с 0, так как если один из префиксов имеет длину 0, то мы знаем наидлиннейшую общую подпоследовательность этих префиксов: это пустая строка. Иными словами, $l[0,j] = l[i,0] = 0$ для всех значений i и j. Когда i и j положительны, мы находим $l[i,j]$, рассматривая меньшие значения i и/или j.

- Если i и j положительны и x_i совпадает с y_j, то $l[i,j] = l[i-1,j-1]+1$.
- Если i и j положительны и x_i отличается от y_j, то $l[i,j]$ равно наибольшему из значений $l[i,j-1]$ и $l[i-1,j]$.

Значения $l[i,j]$ можно рассматривать как записи в таблицу. Нам необходимо вычислять эти значения в возрастающем порядке индексов i и j. Вот как выглядит таблица $l[i,j]$ для нашего примера (что собой представляют затененные клетки, вы узнаете немного позже).

		j	0	1	2	3	4	5	6	7	8	9
		y_j		G	T	A	C	C	G	T	C	A
i	x_i	$l[i,j]$										
0			0	0	0	0	0	0	0	0	0	0
1	C		0	0	0	0	1	1	1	1	1	1
2	A		0	0	0	1	1	1	1	1	1	2
3	T		0	0	1	1	1	1	1	2	2	2
4	C		0	0	1	1	2	2	2	2	3	3
5	G		0	1	1	1	2	2	3	3	3	3
6	A		0	1	1	2	2	2	3	3	3	4

Например, $l[5,8]=3$, что означает, что наидлиннейшая общая подпоследовательность $X_5 =$ CATCG и $Y_8 =$ GTACCGTC имеет длину 3, как мы уже видели на с. 121.

Для того чтобы вычислить значения таблицы в порядке возрастания индексов, перед тем как вычислять определенное значение $l[i,j]$, где i и j положительны, нам необходимо вычислить записи $l[i,j-1]$ (непосредственно слева от $l[i,j]$), $l[i-1,j]$ (непосредственно над $l[i,j]$) и $l[i-1,j-1]$ (вверху слева от $l[i,j]$)[1]. Записи таблицы легко вычислить, идя построчно слева направо или постолбцово сверху вниз.

Приведенная далее процедура обрабатывает таблицу как двумерный массив $l[0..m,0..n]$. После заполнения крайнего слева столбца и крайней сверху строки нулями она заполняет оставшуюся часть массива построчно.

Процедура COMPUTE-LCS-TABLE(X,Y)

Вход: X и Y: две строки длиной m и n соответственно.

Выход: массив $l[0..m,0..n]$. Значение $l[m,n]$ представляет собой длину наидлиннейшей общей подпоследовательности X и Y.

1. Пусть $l[0..m,0..n]$ представляет собой новый массив.
2. Для $i=0$ до m:

   A. Установить $l[i,0]=0$.
3. Для $j=0$ до n:

   A. Установить $l[0,j]=0$.

[1] *Упоминание* $l[i-1,j-1]$ *избыточно, так как это значение мы должны будем вычислить до* $l[i,j-1]$ *и* $l[i-1,j]$.

4. Для $i = 1$ до m:

   A. Для $j = 1$ до n:

      i. Если x_i совпадает с y_j, то установить $l[i, j] = l[i-1, j-1] + 1$.

      ii. В противном случае (x_i отличается от y_j) установить $l[i, j]$ равным большему из значений $l[i, j-1]$ и $l[i-1, j]$. Если $l[i, j-1] = l[i-1, j]$, конкретный выбор не имеет значения.

5. Вернуть массив l.

Поскольку заполнение каждой записи таблицы выполняется за константное время, а таблица содержит $(m+1)\cdot(n+1)$ записей, время работы процедуры Compute-LCS-Table равно $\Theta(mn)$.

Хорошая новость заключается в том, что, как только мы вычислим таблицу $l[i, j]$, ее элемент в правом нижнем углу $l[m, n]$ даст нам длину наидлиннейшей общей подпоследовательности X и Y. Плохая новость заключается в том, что ни одна запись в таблице не говорит нам о том, какие именно символы входят в LCS. Для построения наидлиннейшей общей подпоследовательности за время $O(m+n)$ можно воспользоваться таблицей и строками X и Y. Каким образом было получено значение $l[i, j]$, можно определить, исходя из самого значения $l[i, j]$ и значений, от которых оно зависит: x_i, y_j, $l[i-1, j-1]$, $l[i, j-1]$ и $l[i-1, j]$.

Я предпочитаю рекурсивный вариант этой процедуры, в котором мы собираем наидлиннейшую общую подпоследовательность с конца к началу. Когда эта рекурсивная процедура находит в X и Y одинаковые символы, она добавляет этот символ к концу строящейся наидлиннейшей общей подпоследовательности. Первоначальный вызов данной процедуры выглядит как Assemble-LCS(X,Y,l,m,n).

Процедура Assemble-LCS(X,Y,l,i,j)

Вход:

- X и Y: две строки.
- l: массив, заполненный процедурой Compute-LCS-Table.
- i и j: индексы как в строках X и Y, так и в массиве l.

Выход: наидлиннейшая общая подпоследовательность X_i и Y_j.

1. Если $l[i, j] = 0$, вернуть пустую строку.
2. В противном случае (поскольку $l[i, j]$ положительно и i и j положительны), если x_i совпадает с y_j, вернуть строку, образованную рекурсивным вызовом Assemble-LCS($X,Y,l,i-1,j-1$) с добавлением к ней символа x_i (или y_j).
3. В противном случае ($x_i \neq y_j$), если $l[i, j-1]$ больше, чем $l[i-1, j]$, вернуть строку, образованную рекурсивным вызовом Assemble-LCS($X,Y,l,i,j-1$).
4. В противном случае ($x_i \neq y_j$ и $l[i, j-1]$ не больше $l[i-1, j]$) вернуть строку, образованную рекурсивным вызовом Assemble-LCS($X,Y,l,i-1,j$).

В таблице на с. 122 заштрихованные записи $l[i,j]$ — те, которые рекурсивно посещаются начальным вызовом ASSEMBLE-LCS(X,Y,l,6,9), а зашифрованные символы x_i — те, которые добавляются к строящейся наидлиннейшей общей подпоследовательности. Чтобы понять, как работает процедура ASSEMBLE-LCS, начнем с $i=6$ и $j=9$. При этом мы находим, что и x_6, и y_9 оба равны A. Таким образом, A является последним символом наидлиннейшей общей подпоследовательности X_6 и Y_9, и переходим к рекурсии на шаге 2. В этом рекурсивном вызове $i=5$ и $j=8$. В этот раз выясняется, что x_5 и y_8 являются различными символами, а также, что $l[5,7]=l[4,8]$, и мы переходим к рекурсии на шаге 4. Теперь в рекурсивном вызове $i=4$ и $j=8$. И так далее, до конца построения. Если прочесть заштрихованные символы x_i сверху вниз, то вы получите строку CTCA, которая и представляет собой наидлиннейшую общую подпоследовательность. Если бы в случаях, когда надо делать выбор между $l[i,j-1]$ и $l[i-1,j]$, мы предпочитали переход влево (шаг 3) переходу вверх (шаг 4), то получили бы наидлиннейшую общую подпоследовательность TCGA.

Что касается времени работы процедуры ASSEMBLE-LCS, равного $O(m+n)$, то заметим, что в каждом рекурсивном вызове происходит уменьшение на единицу либо значения i, либо значения j, либо обоих одновременно. После $m+n$ рекурсивных вызовов один из индексов гарантированно станет нулевым, и рекурсия исчерпается на шаге 1.

Преобразование одной строки в другую

Теперь давайте посмотрим, как преобразовать одну строку X в другую строку Y. Мы начнем со строки X и будем конвертировать ее в Y символ за символом. Мы предполагаем, что X и Y состоят из m и n символов соответственно. Как и прежде, будем обозначать i-й символ каждой строки, используя строчное имя строки с индексом i, так что i-й символ строки X обозначается как x_i, а j-й символ строки Y — как y_j.

Чтобы преобразовать X в Y, мы будем строить строку (назовем ее Z) таким образом, чтобы по окончании работы строки Z и Y совпадали. Мы поддерживаем индексы i в строке X и j в строке Z. Мы допускаем выполнение последовательности определенных операций преобразования, которые могут менять строку Z и указанные индексы. Мы начинаем с i и j, равных 1, и в процессе работы мы должны исследовать каждый символ строки X, что означает, что мы остановимся только тогда, когда i достигнет значения $m+1$.

Вот операции, которые мы рассматриваем.

- **Копирование** символа x_i из X в Z путем присвоения z_j значения x_i с последующим увеличением i и j на единицу.
- **Замена** символа x_i из X другим символом a путем присвоения z_j значения a с последующим увеличением i и j на единицу.
- **Удаление** символа x_i из X путем увеличения i на единицу; значение j при этом не изменяется.
- **Вставка** символа a в Z путем присвоения z_j значения a с последующим увеличением j на единицу; значение i при этом не изменяется.

Возможны и другие операции — например, обмен местами двух соседних символов или удаление символов с x_i по x_m как единая операция, — но здесь мы рассматриваем только операции *копирования*, *замены*, *удаления* и *вставки*.

В качестве примера вот последовательность операций, преобразующих строку ATGATCGGCAT в строку CAATGTGAATC (заштрихованные символы представляют собой x_i и z_j после каждой операции).

Операция	X	Z
Исходные строки	ATGATCGGCAT	
Удаление A	ATGATCGGCAT	
Замена T на C	ATGATCGGCAT	C
Замена G на A	ATGATCGGCAT	CA
Копирование A	ATGATCGGCAT	CAA
Копирование T	ATGATCGGCAT	CAAT
Замена C на G	ATGATCGGCAT	CAATG
Замена G на T	ATGATCGGCAT	CAATGT
Копирование G	ATGATCGGCAT	CAATGTG
Замена C на A	ATGATCGGCAT	CAATGTGA
Копирование A	ATGATCGGCAT	CAATGTGAA
Копирование T	ATGATCGGCAT	CAATGTGAAT
Вставка C	ATGATCGGCAT	CAATGTGAATC

Возможны и другие последовательности операций. Например, можно просто по очереди удалять каждый символ из X, а затем вставлять каждый символ из Y в Z.

Каждая из операций преобразования имеет стоимость, которая представляет собой константу, зависящую только от типа операции, но не от участвующих в ней символов. Наша цель — найти последовательность операций, которая преобразует X в Y и имеет минимальную стоимость. Давайте обозначают стоимость операции *копирования* как c_C, стоимость *замены* как c_R, удаления как c_D и вставки как c_I. Для последовательности операций в приведенном выше примере общая стоимость равна $5c_C + 5c_R + c_D + c_I$. Следует считать, что каждая из стоимостей c_C и c_R меньше, чем $c_D + c_I$, потому что в противном случае вместо оплаты c_C для копирования символа или c_R для его замены было бы дешевле заплатить $c_D + c_I$ для удаления символа и вставки его же (вместо копирования) или любого другого (вместо замены).

Зачем вообще нужно преобразовывать одну строку в другую? Одно из приложений связано с вычислительной биологией. Биологам часто надо выяснить, насколько схожи две последовательности ДНК. В одном из способов сравнения двух последовательностей X и Y мы по возможности максимально выстраиваем идентичные символы, вставляя пробелы в обе последовательности (включая пробелы на концах последовательностей) так, чтобы результирующие последовательности, скажем, X' и Y', имели одинаковую длину, но не имели пробелов в одних и тех же позициях. То есть и x'_i, и y'_i одновременно быть пробелами не могут. После такого выравнивания каждой позиции назначается своя оценка:

- -1, если и x'_i, и y'_i одинаковы и не являются пробелами;
- $+1$, если x'_i и y'_i различны и ни один из них не является пробелом;
- $+2$, если x'_i или y'_i представляет собой пробел.

Общая оценка вычисляется как сумма оценок отдельных позиций. Чем она ниже, тем ближе друг к другу две строки. Строки из приведенного выше примера можно выровнять следующим образом (здесь _ обозначает пробел).

```
X': ATGATCG_GCAT_
Y': _CAAT_GTGAATC
    *++--*-*-+--*
```

Знак - под позицией символа указывает на ее оценку, равную -1, знак + — на оценку $+1$, а * — $+2$. В данном конкретном примере общая оценка равна $(6\cdot-1)+(3\cdot1)+(4\cdot2)=5$.

Имеется множество возможных способов вставки пробелов и выравнивания двух последовательностей. Чтобы найти способ, дающий наилучшее соответствие — с минимальной оценкой, — мы используем преобразование со стоимостями $c_C=-1$, $c_R=+1$ и $c_D=c_I=+2$. Чем больше идентичных символов соответствуют друг другу, тем лучше выравнивание, так что отрицательная стоимость операции копирования обеспечивает стимул для увеличения количества таких символов. Пробелы в Y' соответствуют удаленным символам, поэтому в приведенном выше примере первый пробел в Y' соответствует удалению первого символа (A) из X. Пробел в X' соответствует вставке символа, и в приведенном выше примере первый пробел в X' соответствует вставке символа T.

Давайте рассмотрим, как преобразуется строка X в строку Y. Мы будем использовать динамическое программирование с подзадачей вида "преобразовать префиксную строку X_i в префиксную строку Y_j", где i пробегает значения от 0 до m, а j — от 0 до n. Назовем эту подзадачу "задачей $X_i \to Y_j$", так что исходная задача является задачей $X_m \to Y_n$. Обозначим стоимость оптимального решения задачи $X_i \to Y_j$ как $cost[i,j]$. В качестве примера возьмем X = ACAAGC и Y = CCGT; мы будем решать задачу $X_6 \to Y_4$ и при этом воспользуемся следующими стоимостями операций: $c_C=-1$, $c_R=+1$ и $c_D=c_I=+2$. Мы решаем подзадачи вида $X_i \to Y_j$, где i пробегает значения от 0 до 6, а j — от 0 до 4. Например, задача $X_3 \to Y_2$ заключается в преобразовании префиксной строки X_3 = ACA в префиксную строку Y_2 = CC.

Легко определить $cost[i,j]$, когда значение i или j равно нулю, потому что X_0 и Y_0 представляют собой пустые строки. Преобразовать пустую строку в Y_j можно путем j операций вставки, так что $cost[0,j]=j\cdot c_I$. Аналогично преобразовать X_i в пустую строку можно путем i операций удаления, так что $cost[i,0]=i\cdot c_D$. Когда $i=j=0$, пустая строка преобразуется сама в себя, так что очевидно, что $cost[0,0]=0$.

Когда и i, и j положительны, нужно изучить применение оптимальной подструктуры к преобразованию одной строки в другую. Предположим — на минутку — что мы знаем, какой была последняя операция, использовавшаяся для преобразования X_i в Y_j. Это одна из четырех операций: *копирования*, *замены*, *удаления* или *вставки*.

- Если последняя операция была операцией *копирования*, то x_i и y_j должны быть одним и тем же символом. Остающаяся подзадача представляет собой преобразование X_{i-1} в Y_{j-1}, и оптимальное решение задачи $X_i \to Y_j$ должно включать оптимальное решение задачи $X_{i-1} \to Y_{j-1}$. Почему? Потому что если бы мы использовали решение задачи $X_{i-1} \to Y_{j-1}$ со стоимостью, отличной от минимальной, то могли бы использовать решение с минимальной стоимостью и получить лучшее решение $X_i \to Y_j$, чем имеющееся. Таким образом, предполагая, что последняя операция была операцией *копирования*, мы знаем, что $cost[i, j]$ равно $cost[i-1, j-1] + c_C$.

- В нашем примере обратимся к задаче $X_5 \to Y_3$. И x_5, и y_3 представляют собой символ G, так что если последняя операция была копированием G, то, поскольку $c_C = -1$, должно выполняться соотношение $cost[5,3] = cost[4,2] - 1$. Если $cost[4,2] = 4$, то мы должны иметь $cost[5,3] = 3$. Если бы мы могли найти решение задачи $X_4 \to Y_2$ со стоимостью, меньшей 4, то мы могли бы использовать это решение для поиска решения задачи $X_5 \to Y_3$ со стоимостью, меньшей, чем 3.

- Если последняя операция была *заменой*, то с учетом разумного предположения, что мы не заменяем символ им же, символы x_i и y_j должны быть разными. Используя те же рассуждения об оптимальной подструктуре, что и для операции копирования, мы видим, что в предположении, что последняя операция была *заменой*, $cost[i, j] = cost[i-1, j-1] + c_R$.

- В нашем примере рассмотрим задачу $X_5 \to Y_4$. На этот раз x_5 и y_4 представляют собой различные символы (G и T соответственно), так что если последняя операция была *заменой* G на T, то, поскольку $c_R = +1$, должно выполняться соотношение $cost[5,4] = cost[4,3] + 1$. Если $cost[4,3]$ равно 3, то $cost[5,4]$ должно быть равно 4.

- Если последней была операция *удаления*, то у нас нет никаких ограничений на x_i и y_j. Думайте о операции удаления как о пропуске символа x_i и оставлении неизменным префикса Y_j, так что подзадача, которую нужно решить, — это задача $X_{i-1} \to Y_j$. Если предположить, что последняя операция — удаление, то $cost[i, j] = cost[i-1, j] + c_D$.

- В нашем примере рассмотрим задачу $X_6 \to Y_3$. Если последняя операция — *удаление* (удаленным должен быть символ x_6, представляющий собой символ C), то, поскольку $c_D = +2$, мы должны иметь $cost[6,3] = cost[5,3] + 2$. Если $cost[5,3] = 3$, то $cost[6,3]$ должен быть равен 5.

- Наконец, если последняя операция — *вставка*, она оставляет строку X_i нетронутой, но добавляет символ y_j, и решаемая подзадача представляет собой задачу $X_i \to Y_{j-1}$. В предположении, что последняя операция — вставка, мы получаем $cost[i, j] = cost[i, j-1] + c_I$.

- В нашем примере рассмотрим задачу $X_2 \to Y_3$. Если последней операцией была *вставка* (вставленный символ y_3, который представляет собой G), то, поскольку $c_I = +2$, должно выполняться соотношение $cost[2,3] = cost[2,2] + 2$. Если $cost[2,2]$ имеет нулевое значение, то стоимость $cost[2,3]$ должна быть равна 2.

Конечно, мы не знаем заранее, какая из четырех операций использовалась последней. Мы хотим использовать ту, которая дает наименьшее значение $cost[i, j]$. Для заданной комбинации i и j применимы три из четырех операций. Операции *удаления* и *вставки* применимы всегда при любых положительных i и j, а в зависимости от того, одинаковы ли символы x_i и y_j, применима только одна из операций — *копирования* и *замены*. Для вычисления $cost[i, j]$ из других значений стоимости мы определяем, какие три операции возможны, и берем минимальное из значений $cost[i, j]$, даваемых тремя возможными операциями. Другими словами, $cost[i, j]$ представляет собой наименьшее из следующих четырех значений:

- $cost[i-1, j-1] + c_C$, но только если x_i и y_j совпадают;
- $cost[i-1, j-1] + c_R$, но только если x_i и y_j различные;
- $cost[i-1, j] + c_D$;
- $cost[i, j-1] + c_I$.

Так же, как при вычислении наидлиннейшей общей подпоследовательности, мы можем заполнять таблицу *cost* строка за строкой. Это возможно, поскольку, как и в таблице *l*, каждое значение в ячейке $cost[i, j]$, где i и j — положительные, зависит от уже вычисленных значений ячеек таблицы, непосредственно примыкающих к искомой слева, сверху и по диагонали слева сверху.

В дополнение к таблице *cost* мы заполняем таблицу *op*, где $op[i, j]$ дает последнюю операцию, использованную для преобразования X_i в Y_j. Мы можем заполнить запись $op[i, j]$ при заполнении записи $cost[i, j]$. Процедура Compute-Transform-Tables, приведенная ниже, построчно заполняет таблицы *cost* и *op*, рассматривая их как двумерные массивы.

Процедура Compute-Transform-Tables(X, Y, c_C, c_R, c_D, c_I)

Вход:

- X и Y: две строки длиной m и n соответственно;
- c_C, c_R, c_D, c_I: стоимости операций *копирования*, *замены*, *удаления* и *вставки* соответственно.

Выход: массивы $cost[0..m, 0..n]$ и $op[0..m, 0..n]$. Значение $cost[i, j]$ представляет собой минимальную стоимость преобразования префикса X_i в префикс Y_j, так что $cost[m, n]$ является минимальной стоимостью преобразования X в Y. Операция в $op[i, j]$ является последней операцией, выполненной при преобразовании X_i в Y_j.

1. Пусть $cost[0..m, 0..n]$ и $op[0..m, 0..n]$ — новые массивы.
2. Установить $cost[0, 0]$ равным 0.
3. Для $i = 1$ до m:

   A. Установить $cost[i, 0] = i \cdot c_D$ и $op[i, 0] = del\ x_i$.
4. Для $j = 1$ до n:

   A. Установить $cost[0, j] = j \cdot c_I$ и $op[0, j] = ins\ y_j$.

5. Для $i = 1$ до m:

   A. Для $j = 1$ до n:

   (Определяем, какая из операций — копирования или замены — применима, и устанавливаем $cost[i, j]$ и $op[i, j]$ в соответствии с тем, какая из трех применимых операций минимизирует значение $cost[i, j]$.)

   i. Установить $cost[i, j]$ и $op[i, j]$ следующим образом:

      a. Если $x_i = y_j$, установить $cost[i, j] = cost[i-1, j-1] + c_C$ и $op[i, j] = copy\ x_i$.

      b. В противном случае ($x_i \neq y_j$) установить $cost[i, j] = cost[i-1, j-1] + c_R$ и $op[i, j] = rep\ x_i\ by\ y_j$.

   ii. Если $cost[i-1, j] + c_D < cost[i, j]$, установить $cost[i, j] = cost[i-1, j] + c_D$ и $op[i, j] = del\ x_i$.

   iii. Если $cost[i, j-1] + c_I < cost[i, j]$, установить $cost[i, j] = cost[i, j-1] + c_I$ и $op[i, j] = ins\ y_i$.

6. Вернуть массивы *cost* и *op*.

Таблицы *cost* и *op*, вычисленные с помощью процедуры COMPUTE-TRANSFORM-TABLES для нашего примера преобразования строки X = ACAAGC в строку Y = CCGT со стоимостями $c_C = -1$, $c_R = +1$ и $c_D = c_I = +2$, показаны ниже. В строке i и столбце j находятся значения $cost[i, j]$ и $op[i, j]$, причем в последней записи *del* означает удаление символа, *ins* — его вставку, *copy* — копирование, а *rep x by y* — замену символа *x* символом *y*. Например, последняя операция, использованная при преобразовании X_5 = ACAAG в Y_2 = CC, представляет собой замену G на C, а оптимальная последовательность операций по преобразованию ACAAG в CC имеет общую стоимость, равную 6.

	j	0	1	2	3	4
	y_j		C	C	G	T
i	x_i					
0		0	2 *ins* C	4 *ins* C	6 *ins* G	8 *ins* T
1	A	2 *del* A	1 *rep* A *by* C	3 *rep* A *by* C	5 *rep* A *by* G	7 *rep* A *by* T
2	C	4 *del* C	1 *copy* C	0 *copy* C	2 *ins* G	4 *ins* T
3	A	6 *del* A	3 *del* A	2 *rep* A *by* C	1 *rep* A *by* G	3 *rep* A *by* T
4	A	8 *del* A	5 *del* A	4 *rep* A *by* C	3 *rep* A *by* G	2 *rep* A *by* T

5	G	10 *del* G	7 *del* G	6 *rep* G *by* C	3 *copy* G	4 *rep* G *by* T
6	C	12 *del* C	9 *copy* C	6 *copy* C	5 *del* C	4 *rep* C *by* T

Процедура Compute-Transform-Tables заполняет каждую запись таблицы за константное время, так же, как и процедура Compute-Lcs-Table. Поскольку каждая из таблиц содержит $(m+1)\cdot(n+1)$ записей, время работы процедуры Compute-Transform-Tables составляет $\Theta(mn)$.

Для построения последовательности операций, преобразующих строку X в строку Y, мы обращаемся к таблице *op*, начиная с последней записи, $op[m,n]$. Мы рекурсивно, как и в процедуре Assemble-Lcs, добавляем каждую операцию, встречающуюся в таблице *op*, к концу последовательности операций. Выполняющая эти действия процедура Assemble-Transformation приведена ниже. Первоначальный вызов процедуры имеет вид Assemble-Transformation(*op*,*m*,*n*). Последовательность операций для преобразования строки X = ACAAGC в строку Z, совпадающую со строкой Y = CCGT, приведена после описания процедуры Assemble-Transformation.

Процедура Assemble-Transformation(*op*,*i*,*j*)

Вход:

- *op*: таблица операций, заполненная процедурой Compute-Transform-Tables.
- *i* и *j*: индексы таблицы *op*.

Выход: последовательность операций, трансформирующая строку X в строку Y, где X и Y представляют собой входные строки для процедуры Compute-Transform-Tables.

1. Если и i, и j равны нулю, вернуть пустую последовательность.
2. В противном случае (хотя бы одно из значений i и j положительно) выполнить следующие действия.

   A. Если $op[i,j]$ является операцией *копирования* или *замены*, вернуть последовательность, образованную рекурсивным вызовом Assemble-Transformation(*op*, $i-1$, $j-1$), к которой добавлено значение $op[i,j]$.

   B. В противном случае ($op[i,j]$ не является ни операцией *копирования*, ни операцией *замены*), если $op[i,j]$ представляет собой операцию *удаления*, вернуть последовательность, образованную рекурсивным вызовом Assemble-Transformation(*op*, $i-1$,*j*), к которой добавлено значение $op[i,j]$.

   C. В противном случае ($op[i,j]$ не является операцией *копирования*, операцией *замены* или операцией *удаления*, так что $op[i,j]$ должно быть операцией *вставки*) вернуть последовательность, образованную рекурсивным вызовом Assemble-Transformation(*op*,*i*, $j-1$), к которой добавлено значение $op[i,j]$.

Операция	X	Y
Исходные строки	ACAAGC	
Удаление A	ACAAGC	
Копирование C	ACAAGC	C
Удаление A	ACAAGC	C
Замена A на C	ACAAGC	CC
Копирование G	ACAAGC	CCG
Замена C на T	ACAAGC_	CCGT

Так же, как и в случае процедуры Assemble-Lcs, каждый рекурсивный вызов процедуры Assemble-Transformation уменьшает значение i или j (или их оба), а потому рекурсия завершается не более чем за $m+n$ рекурсивных вызовов. Поскольку каждый рекурсивный вызов требует константного времени до и после рекурсии, процедура Assemble-Transformation выполняется за время $O(m+n)$.

В процедуре Assemble-Transformation имеется одна тонкость, требующая более пристального рассмотрения. Рекурсия завершается только по достижении нулевого значения одновременно и i, и j. Предположим теперь, что нулю равно только одно из значений i и j, но не оба одновременно. Каждый из трех случаев на шагах 2A, 2B и 2C приводит к рекурсии с уменьшенным на единицу значением i или j (или обоими значениями одновременно). Не может ли быть выполнен рекурсивный вызов, в котором i или j имеет значение -1? К счастью, ответ на этот вопрос отрицательный. Предположим, что в вызове Assemble-Transformation $j=0$, а i является положительным. В соответствии со способом построения таблицы *op* значение $op[i,0]$ представляет собой операцию *удаления*, так что выполняется шаг 2B. Рекурсивный вызов на шаге 2B вызывает Assemble-Transformation($op,i-1,j$), так что значение j в рекурсивном вызове остается равным нулю. Аналогично, если $i=0$, а j положительно, то $op[0,j]$ является операцией *вставки*, а поэтому выполняется шаг 2C, и в рекурсивном вызове Assemble-Transformation($op,i,j-1$) значение i остается нулевым.

Поиск подстрок

В задаче поиска подстрок у нас есть две строки: ***текстовая строка*** T и ***шаблонная строка*** P. Мы хотим найти все вхождения P в T. Сократим названия строк до "текст" и "шаблон" и предположим, что текст и шаблон состоят из n и m символов соответственно, где $m \le n$ (так как не имеет смысла искать шаблон, больший по размеру, чем текст). Будем обозначать символы в P и T соответственно как $p_1p_2p_3\ldots p_m$ и $t_1t_2t_3\ldots t_n$.

Поскольку мы хотим найти все вхождения шаблона P в текст T, решением будут все величины сдвигов P относительно начала T, где шаблон располагается в тексте. Другими словами, мы говорим, что шаблон P ***встречается в тексте со сдвигом*** s, если подстрока T, которая начинается с t_{s+1}, в точности такая же, как и шаблон P: $t_{s+1}=p_1$, $t_{s+2}=p_2$ и так далее до $t_{s+m}=p_m$. Минимально возможный сдвиг — нулевой, а так как шаблон не должен выходить за пределы текста, максимально возможный сдвиг равен $n-m$. Мы хотим найти

все сдвиги, с которыми P входит в T. Например, если текст $T =$ GTAACAGTAAACG, а шаблон $P =$ AAC, то P встречается в T со сдвигами 2 и 9.

Если мы проверяем, не входит ли шаблон P в текст T с некоторым сдвигом s, нам следует сравнить все m символов P с символами T. Предполагая, что такое сравнение одного символа выполняется за константное время, в наихудшем случае для сравнения всего шаблона потребуется время $\Theta(m)$. Конечно, как только мы найдем несоответствие между символами P и T, нам больше не потребуется проверять остальные символы. Наихудший случай осуществляется при каждой величине сдвига, при которой P встречается в T.

Было бы достаточно легко просто сравнивать шаблон с текстом для каждого возможного сдвига от 0 до $n - m$. Вот как выполняется поиск шаблона AAC в тексте GTAACAGTAAACG для каждого возможного сдвига (совпадающие при сравнении символы затенены).

Величина сдвига	Текст и шаблон	Величина сдвига	Текст и шаблон
0	GTAACAGTAAACG AAC	6	GTAACAGTAAACG AAC
1	GTAACAGTAAACG AAC	7	GTAACAGTAAACG AAC
2	GTAACAGTAAACG AAC	8	GTAACAGTAAACG AAC
3	GTAACAGTAAACG AAC	9	GTAACAGTAAACG AAC
4	GTAACAGTAAACG AAC	10	GTAACAGTAAACG AAC
5	GTAACAGTAAACG AAC		

Увы, этот простой подход довольно неэффективен: при наличии $n - m + 1$ возможных сдвигов, каждый из которых требует для проверки времени $O(m)$, полное время работы алгоритма составляет $O((n - m)m)$. Нам придется проверять почти каждый символ текста m раз.

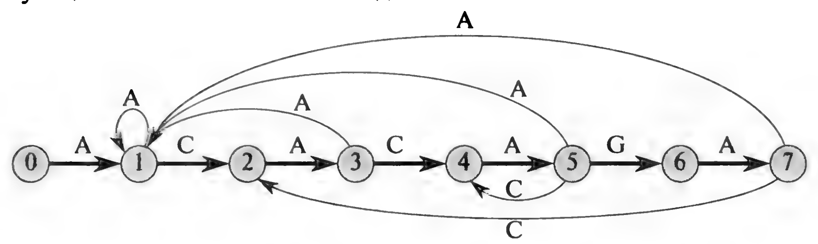
Можно поступить лучше, потому что простой способ сравнения шаблона с текстом для каждого возможного сдвига выдает ценную информацию. В приведенном выше примере, когда мы рассматривали сдвиг $s = 2$, мы просмотрели все символы в подстроке $t_3 t_4 t_5 =$ AAC. Но при следующем сдвиге, $s = 3$, мы опять просматриваем символы t_4 и t_5. Было бы более эффективно, если это возможно, избежать повторного просмотра этих символов. Рассмотрим более интеллектуальный подход к поиску подстрок, который позволяет избежать пустой траты времени, вызванной повторным сканированием текста. Он проверяет каждый символ текста ровно один раз вместо m-кратного их исследования.

Этот более эффективный подход основан на применении ***конечного автомата***. Несмотря на вызывающее название, идея довольно проста. Имеется предостаточно различных приложений конечных автоматов, но здесь мы остановимся только на их применении для поиска подстрок. Конечный автомат, или, для краткости, ***КА***, — это просто набор ***состояний***, а путь от состояния к состоянию основан на последовательности входных

символов. КА начинает работу с определенного состояния и по одному получает входные символы. Основываясь на состоянии, в котором он находится, и полученном символе, конечный автомат переходит в новое состояние.

В нашем приложении поиска подстрок входная последовательность представляет собой символы текста T, и КА будет иметь $m+1$ состояние (на одно больше, чем количество символов в шаблоне P), пронумерованное от 0 до m. (Слово "конечный" в названии "конечный автомат" означает конечное количество состояний автомата.) КА начинает работу из состояния 0. Когда он находится в состоянии k, k последних считанных символов текста соответствуют первым k символам шаблона. Таким образом, всякий раз, когда КА входит в состояние m, он встретил в тексте весь шаблон.

Давайте рассмотрим пример, в котором используются только буквы A, C, G и T. Предположим, что шаблон представляет собой ACACAGA, в котором $m = 7$ символов. Вот соответствующий КА с состояниями от 0 до 7.

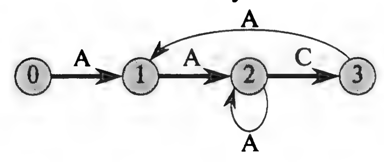


Круги представляют состояния, а помеченные символами стрелки показывают, как КА переходит из одного состояния в другое при получении входных символов. Например, стрелки из состояния 5 помечены как A, C и G. Стрелка в состояние 1, помеченная как A, указывает, что когда КА находится в состоянии 5 и получает символ текста A, он переходит в состояние 1. Аналогично стрелка в состояние 4, помеченная как C, говорит нам, что когда КА находится в состоянии 5 и получает текстовый символ C, он переходит в состояние 4. Обратите внимание на выделенный толстыми стрелками "позвоночник", который при прочтении слева направо дает шаблон ACACAGA. Всякий раз, когда в тексте встречается шаблон, КА перемещается вправо по одному состоянию для каждого символа, пока не достигнет последнего состояния, где он объявляет, что найдено вхождение шаблона в текст. Обратите также внимание на то, что некоторые стрелки отсутствуют — такие, как стрелки, помеченные T. Если стрелка отсутствует, соответствующий переход ведет в состояние 0.

КА хранит таблицу *next-state*, которая индексируется всеми состояниями и всеми возможными входными символами. Значение *next-state*$[s,a]$ представляет собой номер состояния, в которое перейдет КА, если в настоящее время он находится в состоянии s и получил из текста символ a. Вот как выглядит таблица *next-state* для шаблона ACACAGA.

Состояние	Символ A	C	G	T
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	4	0	0
4	5	0	0	0
5	1	4	6	0
6	7	0	0	0
7	1	2	0	0

КА перемещается на одно состояние вправо для каждого символа, который соответствует шаблону, а для каждого символа, который ему не соответствует, он переходит влево или остается в том же состоянии (*next-state*[1, A] равно 1). Позже мы узнаем, как строить таблицу *next-state*, а пока что рассмотрим работу КА для шаблона AAC и входного текста GTAACAGTAAACG. Вот как выглядит соответствующий КА.



Из приведенной схемы КА легко составить таблицу *next-state*, которая в данном случае имеет следующий вид.

	Символ			
Состояние	A	C	G	T
0	1	0	0	0
1	2	0	0	0
2	2	3	0	0
3	1	0	0	0

А вот как выглядит перемещение КА по состояниям при считывании символов из входного текста.

Состояние	0	0	0	1	2	3	1	0	0	1	2	2	3	0
Символ		G	T	A	A	C	A	G	T	A	A	A	C	G

Серой штриховкой выделены два случая, когда КА достигает состояния 3, поскольку достижение этого состояния говорит о том, что в тексте обнаружено вхождение шаблона AAC.

Ниже приведена процедура FA-STRING-MATCHER, выполняющая поиск подстрок. Предполагается, что таблица *next-state* к моменту вызова процедуры уже построена.

Процедура FA-STRING-MATCHER(*T*,*next-state*,*m*,*n*)

Вход:

- *T*, *n*: строка текста и ее длина.
- *next-state*: таблица переходов между состояниями, построенная для заданного шаблона.
- *m*: длина шаблона. Строки таблицы *next-state* индексированы от 0 до *m*, а столбцы индексированы символами, которые могут встретиться в тексте.

Выход: выводит все величины сдвигов, при которых в тексте встречается искомый шаблон.

1. Установить переменную *state* равной нулю.
2. Для $i = 1$ до *n*:

   A. Установить значение *state* равным *next-state*$[state, t_i]$.

   B. Если $state = m$, вывести сообщение “Шаблон найден со сдвигом ” $i - m$.

Если выполнить процедуру FA-String-Matcher для рассмотренного выше примера, в котором $m = 3$, КА достигнет состояния 3 после обработки символов t_5 и t_{12}. Следовательно, процедура выведет сообщения "Шаблон найден со сдвигом 2" ($2 = 5 - 3$) и "Шаблон найден со сдвигом 9" ($9 = 12 - 3$).

Поскольку каждая итерация цикла на шаге 2 выполняется за константное время и этот цикл выполняет ровно n итераций, легко видеть, что время работы процедуры FA-String-Matcher равно $\Theta(n)$.

Это была легкая часть данного материала. Трудная часть материала состоит в построении таблицы *next-state* конечного автомата для заданного шаблона. Вспомним основную идею.

Когда конечный автомат находится в состоянии k, k последних считанных символов текста соответствуют первым k символам шаблона.

Чтобы конкретизировать эту идею, вернемся к КА на с. 133 для шаблона ACACAGA и подумаем о том, почему *next-state*$[5, \mathrm{C}] = 4$. Если КА перешел в состояние 5, пять последних считанных из текста символов — ACACA, которые можно увидеть, если рассмотреть "позвоночник" КА. Если следующий считанный символ — C, он не соответствует шаблону и КА не попадет в состояние 6. Но КА при этом и не вернется в состояние 0. Почему? Потому что четыре последних считанных символа — ACAC, которые соответствуют первым четырем символам шаблона ACACAGA. Вот почему когда КА находится в состоянии 5 и получает на вход символ C, он переходит в состояние 4: последние четыре считанных символа соответствуют первым четырем символам шаблона.

Мы почти готовы познакомиться с правилом построения таблицы *next-state*, но сначала нам нужна пара определений. Вспомним, что i находится в диапазоне от 0 до m, префикс P_i шаблона P представляет собой подстроку, состоящую из первых i символов P. (Когда $i = 0$, префикс является пустой строкой.) Определим ***суффикс*** шаблона как подстроку символов с конца P. Например, AGA — суффикс шаблона ACACAGA. Определим также ***конкатенацию*** строки X и символа a как новую строку, получающуюся путем добавления a к концу X, и будем обозначать ее как Xa. Например, конкатенацией строки CA с символом T является строка CAT.

Наконец, мы готовы к построению *next-state*$[k, a]$, где k — номер состояния от 0 до m, а a — любой символ, который может появиться в тексте. В состоянии k мы считали из текста префикс P_k, т.е. последние k считанных символов текста совпадают с первыми k символами шаблона. Когда мы считываем следующий символ, скажем, a, мы считываем из текста строку $P_k a$ (конкатенация P_k с a). Какой длины префикс P считан нами в этот момент? Или, если переформулировать вопрос, какой длины префикс P находится в конце $P_k a$? Эта длина и является номером следующего состояния.

Более лаконично:

Возьмем префикс P_k (первые k символов шаблона P), выполним его конкатенацию с символом a. Обозначим полученную в результате строку как $P_k a$. Найдем наидлиннейший префикс P, который одновременно является суффиксом $P_k a$. Тогда *next-state*$[k, a]$ представляет собой длину этого наидлиннейшего префикса.

Так как имеется несколько префиксов и суффиксов, давайте посмотрим, как мы определяем, что для шаблона P = ACACAGA *next-state*$[5, \text{C}] = 4$. Поскольку в этом случае $k = 5$, мы берем префикс P_5, который представляет собой ACACA, и добавляем к нему символ C, что дает нам строку ACACAC. Мы хотим найти наидлиннейший префикс ACACAGA, который одновременно является суффиксом ACACAC. Поскольку строка ACACAC имеет длину 6, а суффикс не может быть длиннее строки, можно начать с P_6 и переходить ко все меньшим и меньшим префиксам. В нашем случае P_6 = ACACAG, и это не суффикс ACACAC. Так что теперь мы переходим к префиксу P_5, который представляет собой строку ACACA и также не является суффиксом ACACAC. Далее мы рассматриваем P_4 = ACAC, но теперь этот префикс является суффиксом ACACAC, так что мы останавливаемся, определив, что *next-state*$[5, \text{C}] = 4$.

Вы можете спросить, всегда ли мы можем найти префикс P, который также является суффиксом $P_k a$? Ответ — да, поскольку пустая строка является префиксом и суффиксом любой строки. Когда наидлиннейший префикс P, который одновременно является суффиксом $P_k a$, оказывается пустой строкой, мы устанавливаем *next-state*$[k, a] = 0$. По-прежнему работая с шаблоном P = ACACAGA, давайте посмотрим, как определяется значение *next-state*$[3, \text{G}]$. Конкатенация P_3 с G дает строку ACAG. Мы работаем с префиксами P, начиная с P_4 (так как длина ACAG равна 4), и движемся к меньшим префиксам. Ни один из префиксов ACAC, ACA, AC и A не является суффиксом ACAG, так что мы выясняем, что искомым наидлиннейшим префиксом является пустая строка, а так как она имеет нулевую длину, мы устанавливаем значение *next-state*$[3, \text{G}]$ равным нулю.

Сколько требуется времени для заполнения всей таблицы *next-state*? Мы знаем, что она имеет по одной строке для каждого состояния КА, так что она имеет $m+1$ строк, пронумерованных от 0 до m. Количество столбцов зависит от количества символов, которые могут встретиться в тексте. Назовем это число q, так что таблица *next-state* содержит $q(m+1)$ записей. Для заполнения записи *next-state*$[k, a]$ мы выполняем следующие действия.

1. Образуем строку $P_k a$.
2. Устанавливаем i равным меньшему из значений $k+1$ (длина $P_k a$) и m (длина P).
3. Пока P_i не является суффиксом $P_k a$, выполняем следующее действие:

   A. Устанавливаем i равным $i-1$.

Заранее нам неизвестно, сколько итераций выполнит цикл на шаге 3, но мы знаем, что он сделает их не более чем $m+1$. Мы также не знаем заранее, сколько символов P_i и $P_k a$ должно проверяться в ходе проверки на шаге 3, но мы знаем, что это число всегда не более m. Поскольку цикл выполняет не более $m+1$ итераций и каждая итерация проверяет не более m символов, заполнение одной записи *next-state*$[k, a]$ занимает время $O(m^2)$. Поскольку всего таблица *next-state* содержит $q(m+1)$ записей, общее время ее заполнения составляет $O(m^3 q)$.

На практике заполнение таблицы *next-state* — не столь уж длительная работа. Я закодировал алгоритм поиска подстрок на C++ на моем MacBook Pro с процессором 2.4 ГГц

и скомпилировал его с использованием оптимизации уровня `-O3`. Я передал программе в качестве шаблона строку `a man, a plan, a canal, panama`; алфавитом при этом был набор символов ASCII размером 128. Программа построила таблицу *next-state* с 31 строкой и 127 столбцами (я пропустил столбец для нулевого символа) приблизительно за 1.35 мс. С более коротким шаблоном программа справляется еще быстрее: чтобы построить таблицу для шаблона `panama`, потребовалось около 0.07 мс.

Тем не менее некоторые приложения очень часто прибегают к поискам подстрок, так что время построения таблицы переходов $O(m^3q)$ может создать проблему. Я не буду вдаваться в подробности, но есть способ сократить время до $\Theta(mq)$. В действительности можно сделать еще лучше. Алгоритм KMP (разработанный Кнутом (Knuth), Моррисом (Morris) и Праттом (Pratt)) использует конечный автомат, но при этом вообще избегает создания и заполнения таблицы *next-state*. Вместо этого он использует массив *move-to* для m состояний, который позволяет КА эмулировать наличие таблицы *next-state* и при этом требует для заполнения массива *next-state* время $\Theta(m)$. Хотя это и было сложнее, чем предыдущее задание, но я скомпилировал и протестировал алгоритм KMP на моем MacBook Pro, и для шаблона `a man, a plan, a canal, panama` потребовалось около одной микросекунды для создания массива *move-to*. Для более короткого шаблона `panama` хватило около 600 нс (0.0000006 с). Совсем неплохо! Как и процедура FA-STRING-MATCHER, алгоритм KMP при наличии построенного массива *move-to* выполняет поиск шаблона в тексте за время $\Theta(n)$.

Дальнейшее чтение

В главе 15 CLRS [4] подробно рассматривается динамическое программирование, включая задачу поиска наидлиннейшей общей подпоследовательности. Приведенный в этой главе алгоритм преобразования одной строки в другую представляет собой часть решения задачи, предлагаемой в главе 15 в CLRS. (Задача в CLRS включает две операции, не рассмотренные здесь, — обмен соседних символов местами и удаление суффикса X. Вы же не думаете, что я могу так подвести своих соавторов, что приведу здесь полное решение задачи?)

Алгоритмы поиска подстрок рассматриваются в главе 32 CLRS. В ней приведен алгоритм, основанный на применении конечных автоматов, а также детально рассмотрен алгоритм KMP. В первом издании *Алгоритмы. Построение и анализ* [3] описан алгоритм Бойера–Мура (Boyer–Moore), который особенно эффективен для длинных шаблонов и большого количества символов в алфавите.

8... Основы криптографии

Покупая что-либо через Интернет, вы обычно указываете номер кредитной карты на веб-сайте продавца или на веб-сайте сторонней службы приема оплат. Чтобы сервер получил номер вашей кредитной карты, его приходится отправлять через Интернет. Интернет — открытая сеть, и нет никаких сложностей в перехвате идущей в ней информации. Таким образом, если номер вашей кредитной карты передается через Интернет, не будучи каким-то образом замаскированным, то его могут перехватить и использовать для приобретения товаров и услуг за ваш счет.

Конечно, маловероятно, что кто-то сидит и ждет именно *вашей* отправки через Интернет чего-то, что выглядит как номер кредитной карты. Куда более вероятно, что кто-то ждет такого действия от *кого угодно*, а *вы* можете оказаться его невезучей жертвой. Для вас было бы гораздо безопаснее скрывать номер кредитной карты всякий раз при отправке его через Интернет. Что вы, вероятно, и делаете. Если вы используете защищенный веб-сайт — URL-адрес которого начинается с "`https:`" вместо обычного "`http:`", — то ваш браузер утаивает информацию от посторонних глаз, отправляя ее с помощью процесса под названием *шифрование*. (Протокол https также предоставляет возможность аутентификации, т.е. вы точно знаете, что подключаетесь именно к тому сайту, к которому надо.) В этой главе мы познакомимся поближе с процессом шифрования, а также с обратным к нему процессом *расшифровки*, в котором зашифрованная информация приводится к первоначальному виду. Процессы шифрования и расшифровки образуют фундамент криптографии.

Хотя я, конечно же, считаю номер моей кредитной карты очень важной информацией, требующей защиты, я признаю также, что в мире имеются и более важные вещи. Если кто-то украдет мой номер кредитной карты, национальная безопасность не подвергнется риску. Но если кто-то узнает правду о том, как госдеп инструктирует своих дипломатов, национальной безопасности или по меньшей мере престижу страны может быть нанесен урон. Таким образом, нужны не просто способы шифрования и расшифровки информации, но способы, которые должно быть очень трудно раскрыть.

В этой главе мы рассмотрим некоторые основные идеи, лежащие в основе шифрования и расшифровки. Современная криптография выходит далеко за рамки представленного здесь материала. Не пытайтесь разработать безопасную систему исключительно на основе настоящей главы! Чтобы создать систему, безопасную в теории и на практике, вам нужно гораздо лучше разбираться в современной криптографии. Так, вам нужно следовать установленным стандартам, подобным опубликованным Национальным институтом стандартов и технологий (NIST). Как писал мне Рон Ривест (Ron Rivest) (один из изобретателей криптосистемы RSA, о которой мы поговорим далее в этой главе), "криптография сродни боевым искусствам, и для ее применения на практике нужно знать последние ее достижения". Здесь вы вкратце познакомитесь с некоторыми алгоритмами, продиктованными необходимостью шифрования и расшифровки информации.

В криптографии мы называем исходную информацию ***текстом*** (plaintext), а зашифрованную его версию — ***шифровкой*** (ciphertext). Шифрование, таким образом, преобразует текст в шифровку, а расшифровка преобразует шифровку обратно в исходный текст. Информация, необходимая для выполнения преобразований, известна как криптографический ***ключ***.

Простые подстановочные шифры

В ***простом подстановочном шифре*** текст шифруется путем простой замены одной буквы другой, а расшифровывается — путем обратной замены. Юлий Цезарь переписывался со своими генералами, используя ***сдвиговый шифр***, в котором отправитель заменяет каждую букву сообщения буквой, находящейся на три позиции далее в алфавите, с циклическим переходом в начало по достижении конца алфавита. В 26-буквенном английском алфавите, например, *A* будет заменена буквой *D*, а *Y* — буквой *B* (после *Y* идет *Z*, затем *A* и *B*). При использовании шифра Цезаря зашифрованное сообщение "пришлите мне еще солдат" выглядит как "тулыолхи при иьи фсозгх". По получении такого текста надо заменить каждую его букву той, которая стоит в алфавите на три позиции ранее, с циклическим переходом в конец по достижении начала алфавита. (Само собой разумеется, Цезарь пользовался латинским алфавитом.)

Если вы перехватили зашифрованное таким образом сообщение, и знаете, что оно зашифровано сдвиговым шифром, расшифровать его до смешного легко, даже если вы не знаете ключ (который в этом случае представляет собой величину сдвига): просто пробуйте все возможные значения сдвига до тех пор, пока расшифрованный текст не станет иметь смысл, превратившись в обычный. В случае 26-символьного алфавита требуется проверить не более 25 разных сдвигов.

Сделать шифр немного более безопасным можно путем некоторого взаимно однозначного преобразования каждого символа в некоторый другой, не обязательно расположенный в алфавите на фиксированном расстоянии. Иными словами, можно создать перестановку символов, которая будет использоваться в качестве ключа. Так мы получим шифр, который по-прежнему представляет собой простой шифр, но он будет более сложным, чем сдвиговый. Если в алфавите имеется n символов, то перехвативший сообщение шпион должен выяснить, какую из $n!$ (факториал n) перестановок вы использовали. Факториал — очень быстро растущая с ростом n функция, которая растет быстрее, чем даже экспоненциальная функция.

Так почему бы не ограничиться таким преобразованием каждого символа в другой? Дело в том, что зачастую можно использовать частоты появления букв и их комбинаций для того, чтобы сузить выбор ключа. Рассмотрим сообщение на обычном английском языке — "Send me a hundred more soldiers". Предположим, что при шифровании оно превращается в "Krcz sr h byczxrz sfxr kfjzgrxk". В зашифрованном тексте наиболее часто появляется буква *r*, так что можно (вполне правильно) предположить, что соответствующий символ текста — *e*, наиболее часто встречающийся символ в английском тексте. Затем можно обратить внимание на двухбуквенное слово *sr* в зашифрованном тексте и предположить,

что исходный символ, соответствующий *s*, должен быть одним из *b*, *h*, *m* или *w*, так как в английском языке есть только четыре двухбуквенных слова, оканчивающихся на *e* — *be*, *he*, *me* и *we*. Можно также определить, что символу *h* соответствует исходный символ *a*, потому это единственное однобуквенное слово в нижнем регистре в английском языке.

Конечно, если вы шифруете номера кредитных карт, то можно не слишком беспокоиться о частотах букв или их комбинаций. Но десять цифр дают только 10! уникальных способов преобразования одной цифры в другую, т.е. всего 3 628 800 вариантов. Для компьютера это не слишком много, особенно по сравнению с 10^{16} возможных номеров кредитных карт (16 десятичных цифр), так что злоумышленник, в принципе, может автоматизировать попытки произвести покупки для каждого из 10! способов.

Вы, возможно, обратили внимание на еще одну проблему простых подстановочных шифров: отправитель и получатель должны согласовать ключ. Кроме того, если вы отправляете сообщения различным получателям и не хотите, чтобы каждый из них мог расшифровывать сообщения, предназначенные другим, вы должны создавать отдельные ключи для каждого из получателей.

Криптография с симметричным ключом

Криптография с симметричным ключом подразумевает, что отправитель и получатель используют один и тот же ключ, о котором они должны каким-то образом заранее договориться.

Одноразовые блокноты

Если предположить, что использование симметричного ключа вас устраивает, но простой подстановочный шифр не достаточно безопасен, можно воспользоваться еще одним вариантом — одноразовым блокнотом. Одноразовый блокнот работает с битами. Как вы, возможно, знаете, ***бит*** (bit) — это аббревиатура от "binary digit", двоичная цифра, и он может принимать только два значения — нуль и единица. Цифровые компьютеры хранят информацию в виде последовательности битов. Одни последовательности битов представляют собой числа, другие — символы (с использованием стандартных кодировок типа ASCII или Unicode), а третьи даже являются командами, выполняемыми компьютером.

Одноразовые блокноты применяют к битам операцию ***исключающего или*** (XOR), для обозначения которой мы используем символ $\oplus$.

$$\begin{aligned} 0 \oplus 0 &= 0, \\ 0 \oplus 1 &= 1, \\ 1 \oplus 0 &= 1, \\ 1 \oplus 1 &= 0. \end{aligned}$$

Самый простой способ представления операции XOR — если x является битом, то $x \oplus 0 = x$, а $x \oplus 1$ обращает значение x. Кроме того, если x и y являются битами, то $(x \oplus y) \oplus y = x$: применение исключающего или к x с одним и тем же значением дважды дает x.

Предположим, что я хочу отправить вам однобитовое сообщение. Я могу отправить в качестве шифровки 0 или 1, и мы при этом договариваемся, отправляю ли я вам исходный или инвертированный бит. Если взглянуть на происходящее через призму операции XOR, мы договариваемся о том, с каким битом — нулевым или единичным — я буду выполнять операцию исключающего или с исходным битом. Если вы после этого выполните ту же операцию, что и я, то восстановите исходный текст.

Теперь предположим, что я хочу послать вам двухбитовое сообщение. Я мог оставить оба бита неизменными, инвертировать оба бита или инвертировать только один из битов. И вновь нам надо договориться о том, какие именно биты инвертируются. С точки зрения операции XOR мы должны договориться о значении двухбитового ключа — одного из значений 00, 01, 10 и 11, — с которым будет применена операция исключающего или к исходному тексту. Для расшифровки вы вновь выполняете те же действия, что и я при зашифровывании.

Если исходный текст состоит из b битов — вероятно, из символов ASCII или Unicode, которые в сумме состоят из этих b битов, — то я мог бы сгенерировать случайную последовательность b битов в качестве ключа, а затем побитово применить операцию исключающего или для битов ключа и исходного текста. После получения b-битов шифровки вы могли бы расшифровать ее, точно так же выполнив над ней операцию исключающего или с битами ключа. Эта система называется ***одноразовым блокнотом***[1].

Пока мы выбираем биты ключа случайным образом — этот вопрос будет рассмотрен позже, — подобрать ключ для расшифровки невозможно. Даже если злоумышленник что-то знает об исходном тексте — например, что это текст на английском языке, — для любой шифровки и любого *потенциального* исходного текста существует ключ, преобразующий этот потенциальный исходный текст в данную шифровку. (Это очевидно, поскольку в случае потенциального исходного текста t, шифровки c и ключа k справедливо не только соотношение $t \oplus k = c$, но и $t \oplus c = k$; операция $\oplus$ применяется к t, k и c побитово, т.е. применение исключающего или к i-му биту t и i-му биту k дает i-й бит c.) Таким образом, шифрование с одноразовым блокнотом предотвращает получение злоумышленником любой дополнительной информации об исходном тексте.

Одноразовые блокноты обеспечивают высокую степень безопасности, но ключи должны иметь ту же длину, что и исходный текст, их биты должны выбираться случайным образом, а сами ключи должны иметься у сторон заранее. Как предполагает название, вы должны использовать одноразовый блокнот только один раз. Если вы используете один и

---

[1] *Это название пришло из докомпьютерной эры, когда у каждой стороны был бумажный блокнот, в котором на каждой странице имелись уникальные ключи, причем блокноты у сторон совпадали, т.е. они пользовались одними и теми же последовательностями ключей. Ключ использовался только один раз, после чего соответствующий лист вырывался из блокнота и уничтожался. Такая система использовала сдвиговый шифр, но сдвиги в нем обозначались буквами — от 0, обозначавшегося буквой a, до 25 (буква z). Например, поскольку z означает сдвиг на 25 позиций, m — на 12, а n — на 13, ключ zmn превращает слово dog в слово cat. В отличие от системы, построенной на операции исключающего или, повторное применение ключа не дает исходный текст — расшифровку надо производить в обратном направлении.*

тот же ключ k для двух текстов t_1 и t_2, то $(t_1 \oplus k) \oplus (t_2 \oplus k) = t_1 \oplus t_2$, что может показать, в каких местах два исходных текста имеют одинаковые биты.

Блочные шифры и цепочки

Если исходный текст длинный, ключ в одноразовом блокноте должен иметь ту же длину, так что он может быть довольно громоздким и неудобным. Поэтому некоторые системы с симметричным ключом объединяют два метода: используют короткий ключ и разбивают текст на несколько блоков, применяя ключ поочередно к каждому блоку. То есть они рассматривают исходный текст как состоящий из l блоков $t_1, t_2, t_3, \ldots, t_l$, и шифруют эти блоки текста в l блоков $c_1, c_2, c_3, \ldots, c_l$ зашифрованного текста. Такая система называется ***блочным шифром***.

На практике блочные шифры выполняют шифрование, несколько более сложное, чем простое применение исключающего или с одноразовым блокнотом. Одна из часто используемых криптосистем с симметричным ключом, AES (Advanced Encryption Standard — расширенный стандарт шифрования), включает в себя блочный шифр. Я не буду вдаваться в подробности AES, скажу только, что она использует сложные методы для разделения и перемешивания блока текста для получения зашифрованного текста. AES использует размер ключа 128, 192 или 256 бит, и размер блока — 128 бит.

У блочных шифров имеются свои проблемы. Если в исходном тексте дважды встречается один и тот же блок, то в зашифрованном тексте дважды встретится один и тот же блок зашифрованного текста. Одним из способов решения этой проблемы является метод ***цепочек блоков*** шифра. Предположим, что вы хотите отправить мне зашифрованное сообщение. Вы разбиваете исходный текст t на l блоков $t_1, t_2, t_3, \ldots, t_l$ и создаете l блоков $c_1, c_2, c_3, \ldots, c_l$ зашифрованного текста следующим образом. Скажем, вы зашифровываете блок, применяя к нему некоторую функцию E, а расшифровка блока выполняется с помощью некоторой функции D. Сначала вы создаете первый блок шифровки, c_1, как и ожидается: $c_1 = E(t_1)$. Но перед тем как приступить к зашифровке второго блока, вы выполняете операцию исключающего или между шифруемым блоком и уже зашифрованным блоком c_1, так что $c_2 = E(c_1 \oplus t_2)$. Перед зашифровкой третьего блока вы точно так же выполняете операцию исключающего или: $c_3 = E(c_2 \oplus t_3)$. И так далее, так что в общем случае при зашифровке i-го блока сначала выполняется операция исключающего или с зашифрованным $(i-1)$-м блоком, т.е. $c_i = E(c_{i-1} \oplus t_i)$. Эта формула работает даже для вычисления c_1 из t_1, если принять, что фиктивный блок c_0 состоит из одних нулей (поскольку $0 \oplus x = x$). При расшифровке сначала вычисляем $t_1 = D(c_1)$. Зная c_1 и c_2, можно найти t_2, сначала вычисляя $D(c_2)$, которое равно $c_1 \oplus t_2$, а затем применяя к полученному результату операцию исключающего или с c_1. В общем случае мы расшифровываем c_i и получаем t_i как $t_i = D(c_i) \oplus c_{i-1}$. Как и при шифровании, эта схема работает даже для вычисления t_1, если принять, что фиктивный блок c_0 состоит из одних нулей.

Но это еще не конец. Даже при использовании цепочек шифрование одного и того же исходного текста дает один и тот же результат. Поэтому злоумышленник в состоянии отследить повторную отправку одного и того же сообщения, что может представлять цен-

ную для злоумышленника информацию. Одно из решений заключается в том, чтобы начинать не с заполненного нулями фиктивного блока c_0. Вместо этого блок c_0 генерируется случайным образом. Затем он применяется как при шифровании, так и при расшифровке. Такой блок называется ***вектором инициализации***.

Согласование общей информации

Для того чтобы криптография с симметричным ключом была работоспособна, отправитель и получатель должны договориться о ключе. Кроме того, если они используют блочный шифр с цепочками, то им следует также договориться о векторе инициализации. Как вы понимаете, на практике редко получается заранее договориться по этому поводу. Так как же отправитель и получатель могут договориться о ключе и векторе инициализации? Далее в этой главе мы увидим, как использовать гибридную систему для их безопасной передачи.

Криптография с открытым ключом

Очевидно, что для того, чтобы получатель зашифрованного сообщения мог его расшифровать, и получатель, и отправитель должны знать ключ, использованный для зашифровки. Да?

Нет.

В ***криптографии с открытым ключом*** у каждой из сторон есть по два ключа: ***открытый ключ*** и ***секретный ключ***. Я буду описывать криптографию с открытым ключом между двумя адресатами, между вами и мной, при этом мой открытый ключ я обозначу как P, а мой секретный ключ — как S. У вас имеются собственный открытый и секретный ключи. У каждого из прочих участников (если их больше двух) также имеются собственные открытые и секретные ключи.

Секретный ключ действительно секретен, но открытые ключи общедоступны и могут быть известны кому угодно. Они могут даже находиться в некотором централизованном хранилище, что позволит любому узнать открытый ключ любого другого человека. При определенных условиях мы можем использовать любой из ключей для шифрования переписки. Под "определенными условиями" я имею в виду наличие функций, которые используют открытый и секретный ключи либо для зашифровки исходного текста, либо для расшифровки зашифрованного текста. Обозначим функцию, которую я использую с моим открытым ключом, как F_P, а функцию, используемую с моим секретным ключом, — как F_S.

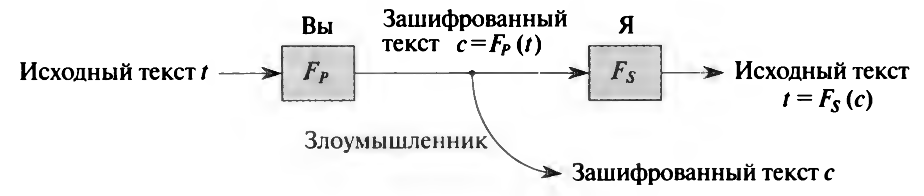
Открытый и секретный ключи связаны соотношением

$$t = F_S\left(F_P(t)\right),$$

так что если вы используете для шифрования мой открытый ключ, а затем я использую для расшифровки мой секретный ключ, то я получу искомый исходный текст. Некоторые другие приложения криптографии с открытым ключом требуют выполнения соотношения $t = F_P\left(F_S(t)\right)$, так что если я зашифрую исходный текст с помощью моего секретного ключа, эту шифровку сможет расшифровать с помощью моего открытого ключа любой.

Для работоспособности криптосистемы любой должен иметь возможность эффективно вычислить мою функцию открытого ключа F_P, но только я должен быть способен вычислить функцию секретного ключа F_S за разумное время. Время, необходимое посторонним для успешного угадывания моей функции F_S без знания моего секретного ключа, должно быть запретительно большим. (Да, здесь я достаточно расплывчато формулирую свои мысли, но скоро вы увидите реальную реализацию криптографии с открытым ключом.) То же самое должно быть справедливо для всех открытых и секретных ключей: функция открытого ключа F_P должна быть эффективно вычислима, но вычислить за разумное время функцию секретного ключа F_S должен только хранитель секретного ключа.

Вот как вы можете послать мне сообщение с использованием криптосистемы с открытым ключом.



Вы начинаете с исходного текста t, находите мой открытый ключ P (возможно, получаете его непосредственно от меня или находите в общедоступном хранилище). Как только вы получаете P, вы шифруете исходный текст с его помощью и получаете зашифрованный текст $c = F_P(t)$, что можно сделать легко и эффективно. Вы отправляете мне зашифрованный текст, так что любой злоумышленник, который перехватывает ваше сообщение мне, видит только зашифрованный текст. Я беру полученный зашифрованный текст c, расшифровываю его, используя мой секретный ключ, и получаю исходный текст $t = F_S(c)$. Кто угодно может легко и быстро зашифровать исходный текст моим открытым ключом, но только я могу расшифровать зашифрованное и воспроизвести исходный текст за разумное время.

На практике нужно убедиться, что функции F_P и F_S корректно работают вместе. Мы хотим, чтобы F_P давала различные зашифрованные тексты для каждого возможного исходного текста. Если предположить, что функция F_P дает один и тот же результат для двух различных исходных текстов t_1 и t_2, т.е. что $F_P(t_1) = F_P(t_2)$, то при попытке расшифровать зашифрованный текст $F_P(t_1)$ с помощью функции F_S я не буду знать, был ли мне передан текст t_1 или t_2. С другой стороны, вполне нормально и даже предпочтительно внесение в процесс шифрования элемента случайности, так что один и тот же исходный текст при шифровании с помощью одного и того же открытого ключа дает всякий раз разные шифровки. (Криптосистема RSA, которую мы рассмотрим немного позже, оказывается гораздо более безопасной, когда исходный текст представляет собой лишь небольшую часть шифруемого сообщения, а остальное тело сообщения является фиктивным случайным “заполнителем”.) Конечно, функция расшифровки F_S должна быть разработана соответствующим образом и давать корректный исходный текст при расшифровке разных шифровок, получающихся при разном заполнении.Однако здесь возникает одна проблема. Исходный текст t может принимать произвольное количество возможных значений — фактически быть произвольной длины, — и количество зашифрованных значений,

в которые функция F_P превращает t, должно быть по крайней мере равно количеству значений, которые может принимать t. Как построить функции F_P и F_S при дополнительном ограничении, что F_P должна быть легко вычислима любым, а F_S — только мною? Это трудно, но выполнимо, если мы ограничим количество возможных исходных текстов, т.е. если используем блочный шифр.

Криптосистема RSA

Криптография с открытым ключом — прекрасная концепция, но она опирается на возможность найти функции F_P и F_S, которые корректно работают вместе, и при этом F_P может быть легко вычислена любым, а F_S — только владельцем секретного ключа. Мы называем такую схему ***криптосистемой с открытым ключом***, а ***криптосистема RSA***, или просто ***RSA***[2], является примером такой схемы.

RSA основана на применении ряда положений теории чисел, многие из которых относятся к модульной арифметике. В модульной арифметике мы выбираем некоторое положительное целое число n, и всякий раз, когда мы достигаем добавлением единиц значения n, мы тут же возвращаемся к нулю. Арифметические действия в модульной арифметике похожи на соответствующие действия в обычной арифметике, но все результаты получаются путем взятия остатка от деления на n. Например, если мы работаем по модулю 5, то единственными возможными значениями являются 0, 1, 2, 3 и 4, и $3+4=2$, поскольку 7 при делении на 5 дает остаток 2. Определив оператор mod для вычисления остатков, мы можем записать, что $7 \bmod 5 = 2$. Еще одно представление модульной арифметики — арифметика на часах, у которых на циферблате 12 заменено нулем. Если вы идете спать в 11 и спите 8 часов, то вы проснетесь в 7 часов: $(11+8) \bmod 12 = 7$.

Самое приятное в модульной арифметике заключается в том, что мы можем применить операцию mod посреди вычислений, и это не приведет к изменению результата[3]:

$$\begin{aligned}
(a+b) \bmod n &= ((a \bmod n)+(b \bmod n)) \bmod n, \\
ab \bmod n &= ((a \bmod n)(b \bmod n)) \bmod n, \\
a^b \bmod n &= (a \bmod n)^b \bmod n.
\end{aligned}$$

---

[2] *Это название представляет собой аббревиатуру от фамилий изобретателей системы — Рональда Ривеста (Ronald Rivest), Ади Шамира (Adi Shamir) и Леонарда Адельмана (Leonard Adelman).*

[3] *В качестве примера, чтобы увидеть, что* $ab \bmod n = ((a \bmod n)(b \bmod n)) \bmod n$, *предположим, что* $a \bmod n = x$ *и* $b \bmod n = y$. *Тогда существуют целые числа* i *и* j, *такие, что* $a = ni + x$ *и* $b = nj + y$, *и,образом,*

$$\begin{aligned}
ab \bmod n &= (ni + x)(nj + y) \bmod n \\
&= (n^2 ij + xnj + yni + xy) \bmod n \\
&= ((n^2 ij \bmod n) + (xnj \bmod n) + (yni \bmod n) + (xy \bmod n)) \bmod n \\
&= xy \bmod n \\
&= ((a \bmod n)(b \bmod n)) \bmod n.
\end{aligned}$$

Кроме того, для любого целого x справедливо соотношение $xn \bmod n = 0$.

Чтобы выполнить условия, налагаемые на открытый ключ криптосистемы RSA, также должны выполняться два теоретико-числовых свойства, относящиеся к простым числам. Как вы, возможно, знаете, ***простое число*** — это целое число, большее 1, которое имеет только два целочисленных делителя: единицу и само это число. Например, число 7 является простым, но 6 таковым не является, так как его можно разложить на множители как $2 \cdot 3$. Первое свойство, от которого зависит работоспособность RSA, заключается в том, что если у вас имеется произведение двух больших секретных простых чисел, то никто не сможет определить эти числа за разумное время. Напомним, что в главе 1, "Что такое алгоритмы и зачем они нужны", говорилось, что для проверки на простоту можно протестировать все возможные нечетные делители, не превышающие квадратного корня исходного числа; однако если это число очень большое — скажем, состоит из сотен или тысяч цифр, — то его квадратный корень, который состоит из половинного количества цифр, тоже представляет собой очень большое число. Хотя *теоретически* так можно было бы найти один из множителей, необходимые для этого ресурсы (время и/или вычислительные мощности) делают поиск множителя невозможным на практике[4].

Второе свойство заключается в том, что, хотя разложить большое простое число на множители очень трудно, совсем нетрудно определить, является ли большое число простым. Вы можете думать, что невозможно определить, что число является составным и при этом не найти хотя бы один его нетривиальный сомножитель (который не является ни единицей, ни самим этим числом), но в действительности это можно сделать. Одним из способов является проверка на простоту AKS[5], первый алгоритм для проверки простоты n-битового числа за время $O(n^c)$ для некоторой константы c. Хотя теоретически метод AKS считается эффективным, с практической точки зрения для больших чисел он непригоден. Вместо него можно воспользоваться проверкой на простоту Миллера–Рабина (Miller–Rabin primality test). Недостатком теста Миллера–Рабина является то, что он может сделать ошибку, объявив простым составное число. (Если этот метод объявляет число составным, то оно действительно составное.) Хорошей новостью является то, что ошибки крайне редки — одна на 2^s, где мы можем выбрать любое положительное значение s, какое захотим. Так что если мы готовы мириться с одной ошибкой из, скажем, 2^{50} проверок, то можно *почти* идеально точно выяснить, является ли число простым. Вспомните из главы 1, "Что такое алгоритмы и зачем они нужны", что 2^{50} — это примерно миллион миллиардов, или около 1000000000000000. Если и это вас не устраивает, то ценой немного больших усилий вы можете добиться одной ошибки из 2^{60} проверок, т.е. примерно еще в 1000 раз реже. Время работы теста Миллера–Рабина линейно зависит от параметра s, так что увеличение s на 10, от 50 до 60, увеличивает время работы только на 20%, но при этом снижает вероятность ошибки в 2^{10} раз (что равно 1024).

---

[4] Например, если число имеет 1000 бит, то его квадратный корень имеет 500 бит и может достигать значения 2^{500}. Даже если кто-то сможет проверять триллион триллионов возможных делителей в секунду, то задолго до достижения им значения 2^{500} Солнце погаснет...

5 Аббревиатура от имен первооткрывателей метода — Маниндры Агравала (Manindra Agrawal), Нираджа Кайяла (Neeraj Kayal) и Нитина Саксены (Nitin Saxena).

Вот как вкратце работает криптосистема RSA. Позже мы рассмотрим некоторые вопросы ее функционирования детальнее.

1. Выберем наугад два очень больших простых числа, p и q, которые не равны друг другу. Что такое очень большое число? Состоящее по крайней мере из 1024 бит, или 309 десятичных цифр. Если еще больше — тем лучше.
2. Вычислим $n = pq$. Это число имеет по меньшей мере 2048 бит, или 618 десятичных цифр.
3. Вычислим $r = (p-1)(q-1)$. Это значение почти такое же большое, как и значение n.
4. Выберем небольшое нечетное целое e, ***взаимно простое*** с r, т.е. единственный общий делитель e и r должен быть равен 1. Нам подойдет любое такое небольшое целое число.
5. Вычислим d как ***мультипликативное обратное*** e по модулю r. То есть $ed \bmod r$ должно быть равно 1.
6. Объявим ***открытый ключ RSA*** как пару $P = (e, n)$.
7. Пара $S = (d, n)$ представляет собой ***секретный ключ RSA***, и ее не должен знать никто.
8. Определим функции F_P и F_S как

$$F_P(x) = x^e \bmod n,$$
$$F_S(x) = x^d \bmod n.$$

Эти функции могут работать как с блоком исходного текста, так и с блоком зашифрованного текста, биты которых рассматриваются как представляющие большие целые числа.

Давайте рассмотрим пример, но для лучшего понимания используем небольшие числа.

1. Выбираем простые числа $p = 17$ и $q = 29$.
2. Вычисляем $n = pq = 493$.
3. Вычисляем $r = (p-1)(q-1) = 448$.
4. Выбираем $e = 5$, взаимно простое с 448.
5. Вычисляем $d = 269$. Проверяем: $ed = 5 \cdot 269 = 1345$, так что $ed \bmod r = 1345 \bmod 448 = (3 \cdot 448 + 1) \bmod 448 = 1$.
6. Объявляем открытый ключ RSA $P = (5, 493)$.
7. Сохраняем $S = (269, 493)$ как секретный ключ RSA.
8. В качестве примера вычисляем $F_P(327)$:

$$\begin{aligned} F_P(327) &= 327^5 \bmod 493 \\ &= 3\,738\,856\,210\,407 \bmod 493 \\ &= 259. \end{aligned}$$

Если мы вычислим $F_S(259) = 259^{269} \bmod 493$, то должны получить 327. Мы сделаем это, но у нас нет ни малейшего желания видеть все 650 цифр числа 259^{269}. Их не так

сложно вычислить, например, найдя в Интернете калькулятор с произвольной точностью вычислений. Но, поскольку мы работаем с модульной арифметикой, значение 259^{269} вычислять не требуется. Мы можем выразить все промежуточные результаты по модулю 493, поступив следующим образом. Начнем с единицы, и 269 раз выполним следующие действия: вычислить произведение последнего полученного результата на 259 по модулю 493. Результат будет тем же, что и при явном вычислении 259^{269}, а именно — 327. (Я для этого написал небольшую компьютерную программу.)

А вот те детали работы RSA, о которых я обещал поговорить подробнее.

- Как работать с числами, состоящими из сотен цифр?
- Хотя проверка того, является ли некоторое число простым, несложная, могу ли я быть уверен в том, что найду большие простые числа за разумное время?
- Как мне найти e, взаимно простое с r?
- Как мне вычислить d, мультипликативно обратное e по модулю r?
- Как за разумное время вычислить $x^d \bmod n$, если d велико?
- Как мне убедиться в том, что функции F_P и F_S обратны одна к другой?

Арифметика больших чисел

Очевидно, что настолько большие числа, которые требуются в реализации RSA, не могут поместиться в регистрах большинства компьютеров (типичный размер которых — 64 бита). К счастью, несколько пакетов программного обеспечения и даже некоторые языки программирования — например, Python — позволяют работать с целыми числами, которые не имеют никакого фиксированного предельного размера.

Кроме того, арифметика в RSA — модульная арифметика, которая позволяет нам ограничить размеры целых чисел, с которыми нам приходится работать. Например, при вычислении $x^d \bmod n$ мы вычисляем промежуточные результаты, представляющие собой значение x, возведенное в разные степени, но по модулю n, что означает, что они находятся в диапазоне от 0 до $n-1$. Кроме того, если вы фиксируете максимальные размеры p и q, то вы фиксируете и максимальный размер n, что, в свою очередь, означает, что возможна аппаратная реализация RSA.

Поиск больших простых чисел

Найти большое простое число можно путем неоднократной произвольной генерации большого нечетного числа с последующим применением теста простоты Миллера–Рабина для выяснения, является ли это число простым. Так, среди прочих можно натолкнуться на простое число. Можно решить, что так придется искать большое простое число слишком долго. Что если простые числа с их ростом становятся крайне редки? Придется потратить огромное количество времени на поиск простой иголки в составном стоге сена.

Однако не нужно беспокоиться. ***Теорема о простых числах*** говорит нам о том, что при стремлении m к бесконечности количество простых чисел, не превышающих m, стремится к $m/\ln m$, где $\ln m$ — натуральный логарифм числа m. Если выбрать целое m случайным

образом, то шанс, что оно простое, — около 1 из $\ln m$. Теория вероятностей говорит нам о том, что в среднем придется испытать только около $\ln m$ чисел недалеко от m, прежде чем найдется простое число. Если я ищу простые числа p и q размером 1024 бита, то m равно 2^{1024}, а $\ln m$ приблизительно равен 710. Компьютер может выполнить тест простоты Миллера–Рабина для 710 чисел очень быстро.

На практике можно использовать более простой тест, чем тест Миллера–Рабина. ***Малая теорема Ферма*** утверждает, что если m является простым числом, то $x^{m-1} \bmod m$ равно единице для любого числа x в диапазоне от 1 до $m-1$. Обратное — что если $x^{m-1} \bmod m$ равно единице для любого числа x в диапазоне от 1 до $m-1$, то m является простым — не обязательно справедливо, но среди больших чисел исключения очень редки. Фактически этого почти всегда достаточно, чтобы просто проверять нечетные целые числа m и объявлять m простым, если $2^{m-1} \bmod m$ равно 1. Позже мы увидим, как вычислить $2^{m-1} \bmod m$ с помощью всего лишь $\Theta(\lg m)$ умножений.

Как найти число, взаимно простое с другим числом

Нам нужно найти небольшое нечетное целое число e, взаимно простое с числом r. Два числа являются взаимно простыми, если их наибольший общий делитель равен 1. Будем использовать алгоритм для вычисления наибольшего общего делителя двух целых чисел, который описан еще древнегреческим математиком Евклидом. В теории чисел имеется теорема, которая гласит, что если есть a и b — целые числа, не равные одновременно нулю, то их наибольший общий делитель g равен $ai+bj$ для некоторых чисел i и j. (Кроме того, g — наименьшее число, которое может быть сформировано таким образом, но этот факт сейчас не играет для нас никакой роли.) Один из коэффициентов i и j может быть отрицательным; например, наибольший общий делитель 30 и 18 равен 6, а $6 = 30i + 18j$ при $i = -1$ и $j = 2$.

Ниже алгоритм Евклида приведен в форме, которая дает наибольший общий делитель g чисел a и b, а также коэффициенты i и j. Эти коэффициенты пригодятся чуть позже, когда будет нужно искать мультипликативное обратное к e по модулю p. Если у меня есть значение-кандидат для e, я вызываю Euclid(r,e). Если первый элемент тройки, возвращенной этим вызовом, равен 1, то проверяемое значение-кандидат для e является взаимно простым с r. Если первый элемент представляет собой любое другое число, то r и проверяемое значение-кандидат для e имеют общий делитель, больший, чем 1, так что они не являются взаимно простыми.

Процедура Euclid(a,b)

Вход: a и b: два целых числа.

Выход: тройка (g,i,j), такая, что g является наибольшим общим делителем a и b, и $g = ai + bj$.

1. Если $b = 0$, вернуть тройку $(a,1,0)$.
2. В противном случае ($b \neq 0$) выполнить следующие действия.

A. Рекурсивно вызвать Euclid(b, a mod b) и присвоить полученный результат тройке (g,i',j'), т.е. присвоить g значение первого элемента возвращенной тройки, i' — значение второго элемента возвращенной тройки, а j' — третьего элемента.

B. Присвоить $i = j'$.

C. Присвоить $j = i' - \lfloor a/b \rfloor j'$.

D. Вернуть тройку (g,i,j).

Я не буду разбираться, почему эта процедура корректно работает[6], не буду и анализировать ее время работы; просто сообщу, что при вызове Euclid(r,e) количество рекурсивных вызовов равно $O(\lg e)$. Таким образом, я могу быстро проверить, равен ли наибольший общий делитель чисел r и значения-кандидата для e единице (не забывайте, что e мало). Если это не так, я могу испытать другое значение-кандидат для e, и так далее, пока не найду взаимно простое с r. Сколько в среднем кандидатов мне придется перепробовать? Немного. Если я ограничу мой выбор для e нечетными простыми числами, меньшими, чем r (что легко проверяется тестом Миллера–Рабина или тестом, основанным на малой теорема Ферма), то весьма вероятно, что любой мой выбор будет взаимно простым с r. Это связано с тем, что согласно теореме о простых числах имеется около $r/\ln r$ простых чисел, меньших r, но еще одна теорема показывает, что r не может иметь более чем $\lg r$ простых множителей. Поэтому я вряд ли столкнусь с простым сомножителем r.

Вычисление мультипликативного обратного в модульной арифметике

После того как я получил r и e, мне нужно вычислить d — обратное к e по модулю r, такое, что $ed \bmod r = 1$. Мы уже знаем, что вызов Euclid(r,e) возвращает тройку вида $(1,i,j)$, т.е. наибольшим общим делителем e и r является единица (потому что эти числа взаимно простые) и что $1 = ri + ej$. Поэтому можно просто присвоить $d = j \bmod r$[7]. Дело в том, что мы работаем по модулю r, а потому можем рассматривать обе стороны равенства по модулю r:

---

[6] *Вызов* EUCLID*(0,0) возвратит тройку (0,1,0), так что 0 рассматривается как наибольший общий делитель двух нулей. Однако для нас совершенно неважно, что возвращает вызов* EUCLID*(0,0), так как параметр a в первом вызове будет положительным и останется таковым в любом рекурсивном вызове.*

[7] *Вспомним, что j может быть отрицательным. Один из способов трактовки $j \bmod r$ при отрицательном j и положительном r — начать с j и прибавлять r до тех пор, пока не получится неотрицательное число, которое и будет равно $j \bmod r$. Например, чтобы найти $-27 \bmod 10$, мы работаем с последовательностью $-27, -17, -7, 3$. Получив последнее значение, мы останавливаемся и говорим, что $-27 \bmod 10 = 3$.*

$$\begin{aligned}1 \bmod r &= (ri + ej) \bmod r \\ &= ri \bmod r + ej \bmod r \\ &= 0 + ej \bmod r \\ &= ej \bmod r \\ &= (e \bmod r)\cdot(j \bmod r) \bmod r \\ &= e(j \bmod r) \bmod r.\end{aligned}$$

(Последняя строка следует из того, что $e < r$, откуда вытекает, что $e \bmod r = e$.) Так что мы получаем $1 = e(j \bmod r) \bmod r$, что означает, что можно присвоить d значение j из тройки, возвращенной вызовом EUCLID(r,e), взятое по модулю r. Я использую $j \bmod r$ вместо просто j в случае, когда j выходит за рамки диапазона от 0 до $r-1$.

Как быстро возвести число в целую степень

Хотя e и является небольшим числом, d может быть большим, а нам для вычисления функции F_S требуется вычислять $x^d \bmod n$. Несмотря на то что мы работаем по модулю n, т.е. все промежуточные значения будут находиться в диапазоне от 0 до $n-1$, выполнять умножение чисел d раз не хочется даже по модулю. К счастью, это и не надо делать. Используя метод ***многократного возведения в квадрат***, можно выполнить всего лишь $\Theta(\lg d)$ умножений. Этот же метод можно использовать и при проверке простоты, основанной на малой теорема Ферма.

Идея заключается в следующем. Мы знаем, что d — число неотрицательное. Сначала предположим, что d четно. Тогда $x^d = \left(x^{d/2}\right)^2$. Теперь предположим, что d нечетное — тогда $x^d = \left(x^{(d-1)/2}\right)^2 \cdot x$. Эти наблюдения дают нам красивый рекурсивный способ вычисления x^d, базовый случай которого осуществляется, когда $d = 0$: $x^0 = 1$. Описанная далее процедура воплощает этот подход, выполняя все арифметические операции по модулю n.

Процедура MODULAR-EXPONENTIATION(x,d,n)

Вход: x, d, n: три целых числа, x и d — неотрицательные, а n — положительное.

Выход: возвращает значение $x^d \bmod n$.

1. Если $d = 0$, вернуть 1.
2. В противном случае (d положительно), если d четно, выполнить рекурсивный вызов MODULAR-EXPONENTIATION(x, $d/2$, n), установить z равным результату рекурсивного вызова и вернуть $z^2 \bmod n$.
3. В противном случае (d положительно и нечетно) выполнить рекурсивный вызов MODULAR-EXPONENTIATION(x, $(d-1)/2$, n), установить z равным результату рекурсивного вызова и вернуть $(z^2 \cdot x) \bmod n$.

Параметр d уменьшается по меньшей мере в два раза при каждом рекурсивном вызове. После не более чем $\lfloor \lg d \rfloor + 1$ вызовов d уменьшается до 0 и рекурсия заканчивается. Таким образом, эта процедура выполняет умножение чисел $\Theta(\lg d)$ раза.

Демонстрация того, что функции F_P и F_S обратны одна к другой

Предупреждение. *Нас ждет масса вопросов теории чисел и модульной арифметики. Если вы готовы принять без доказательства тот факт, что функции F_P и F_S обратны одна к другой, пропустите этот подраздел и переходите сразу к разделу "Гибридные криптосистемы".*

Чтобы RSA была криптосистемой с открытым ключом, функции F_P и F_S должны быть обратны одна к другой. Если взять блок текста t, рассматривать его как целое число, меньшее, чем n, и передать его функции F_P, мы получим значение $t^e \bmod n$; если теперь передать этот результат функции F_S, мы получим $\left(t^e\right)^d \bmod n$, что равно $t^{ed} \bmod n$. Если выполнить действия в обратном порядке, мы получим $\left(t^d\right)^e \bmod n$, что, опять же, равно $t^{ed} \bmod n$. Мы должны показать, что для любого блока текста t, рассматриваемого как целое число, меньшее n, справедливо соотношение $t^{ed} \bmod n = t$.

Вот краткое изложение нашего подхода. Напомним, что $n = pq$. Покажем, что $t^{ed} \bmod p = t \bmod p$ и $t^{ed} \bmod q = t \bmod q$. Тогда, используя другой факт теории чисел, мы заключим, что $t^{ed} \bmod pq = t \bmod pq$ — другими словами, что $t^{ed} \bmod n = t \bmod n$, что просто представляет собой t, так как $t < n$.

Нам нужно снова воспользоваться малой теоремой Ферма, и это помогает понять, почему мы принимаем r равным произведению $(p-1)(q-1)$. Так как p — простое число, то, если $t \bmod p \neq 0$, выполняется соотношение $(t \bmod p)^{p-1} \bmod p = 1$.

Вспомним, что мы определили e и d как мультипликативно обратные по модулю r: $ed \bmod r = 1$. Иными словами, $ed = 1 + h(p-1)(q-1)$ для некоторого целого числа h. Если $t \bmod p \neq 0$, то мы получаем

$$\begin{aligned}
t^{ed} \bmod p &= (t \bmod p)^{ed} \bmod p \\
&= (t \bmod p)^{1+h(p-1)(q-1)} \bmod p \\
&= \left((t \bmod p)\cdot\left((t \bmod p)^{p-1} \bmod p\right)^{h(q-1)}\right) \bmod p \\
&= (t \bmod p)\cdot\left(1^{h(q-1)} \bmod p\right) \\
&= t \bmod p.
\end{aligned}$$

Конечно, если $t \bmod p = 0$, то $t^{ed} \bmod p = 0$.

Аналогичные рассуждения показывают, что если $t \bmod q \neq 0$, то $t^{ed} \bmod q = t \bmod q$, а если $t \bmod q = 0$, то $t^{ed} \bmod q = 0$.

Нам нужен еще один факт из теории чисел, чтобы завершить нашу работу: поскольку p и q взаимно простые (они оба — простые числа), если и $x \bmod p = y \bmod p$, и $x \bmod q = y \bmod q$, то $x \bmod pq = y \bmod pq$ (этот факт является следствием китайской теоремы об остатках). Подставим t^{ed} вместо x и t вместо y и, вспомнив, что $n = pq$ и $t < n$, получим $t^{ed} \bmod n = t \bmod n = t$, а это именно то, что и требовалось показать.

Гибридные криптосистемы

Хотя мы можем выполнять арифметические операции с очень большими числами, на практике мы платим за это снижением скорости вычислений. Шифрование и расшифровка длинного сообщения, содержащего сотни или тысячи блоков исходного текста, могут вызвать заметные задержки. RSA часто используется в гибридных системах, которые частично представляют собой системы с открытым ключом и частично — с симметричным ключом.

Вот как вы могли бы прислать мне зашифрованное сообщение в случае применения гибридной системы. Мы согласуем, какие системы с открытым и симметричным ключами будут использоваться; скажем, это RSA и AES. Выберите ключ k для AES и зашифруйте его моим открытым ключом RSA, получив $F_P(k)$. Затем с использованием ключа k вы зашифровываете последовательности блоков открытого текста с помощью AES и получаете блоки зашифрованного текста. После этого вы присылаете мне $F_P(k)$ и зашифрованный текст. Я расшифровываю $F_P(k)$ путем вычисления $F_S(F_P(k))$ и получаю ключ k AES, а затем использую k для расшифровки зашифрованного текста с помощью AES, тем самым восстанавливая исходный текст. Если применяется блочный шифр с цепочками и нужен вектор инициализации, его можно зашифровать как с помощью RSA, так и посредством AES.

Вычисление случайных чисел

Как мы уже видели, некоторые криптосистемы требуют от нас генерации случайных чисел, а точнее — случайных целых положительных чисел. Поскольку мы представляем целое число как последовательность битов, нам нужен способ случайной генерации битов, которые затем рассматриваются как представляющие целое число.

Случайные биты могут поступать только от действительно случайных процессов. Но как может программа, выполняемая на компьютере, быть случайным процессом? В большинстве случаев компьютерная программа — это точно определенный набор детерминированных инструкций, которые для одних и тех же входных данных всегда дают один и тот же результат. Для поддержки криптографического программного обеспечения некоторые современные процессоры снабжены командами, которые генерируют случайные биты на основе случайных процессов, например на тепловом шуме в схеме. Дизайнеры этих процессоров сталкиваются с тройной проблемой: генерировать биты надо достаточно быстро, при этом обеспечивая соответствие основным статистическим критериям случайности, а кроме того, потреблять разумное количество энергии в процессе создания и тестирования случайных битов.

Обычно криптографические программы получают биты от ***генератора псевдослучайных чисел***, или ГПСЧ. ГПСЧ — детерминированная программа, которая производит последовательность значений, основанных на некотором начальном значении, в соответствии с детерминированным правилом, встроенным в программу, которая выдает очередное псевдослучайное значение на основании текущего. Если вы начинаете работу ГПСЧ с одного и того же начального значения, вы всякий раз будете получать одну и ту же после-

довательность значений. Такое повторяемое поведение хорошо для отладки, но не годится для криптографии. Последние стандарты генераторов случайных чисел для криптосистем требуют определенных реализаций ГПСЧ.

Если вы используете ГПСЧ для генерации битов, выглядящих случайными, желательно каждый раз начинать с нового начального значения, которое должно быть случайным. В частности, оно должно основываться на битах несмещенных (равновероятных 0 и 1), независимых (независимо от информации о предыдущих сгенерированных битах шанс правильно угадать следующий бит равен 50%) и непредсказуемых для злоумышленника, который пытается разрушить вашу криптосистему. Если у вашего процессора есть команда генерации случайных битов, это наилучший способ генерации исходного значения для ГПСЧ.

Дальнейшее чтение

Криптография является лишь одним из компонентов безопасности компьютерных систем. Книга Смита (Smith) и Марчесини (Marchesini) [20] охватывает вопросы компьютерной безопасности весьма широко, включая криптографию и способы атак криптосистем.

Если вы хотите углубиться в вопросы криптографии, я рекомендую книги Каца (Katz) и Линделла (Lindell) [9] и Менезеса (Menezes), ван Ооршота (van Oorschot) и Ванстоуна (Vanstone) [16]. Глава 31 CLRS [4] содержит беглый обзор основ теории чисел, на которых базируется криптография, а также описание RSA и тест на простоту Миллера–Рабина. Диффи (Diffie) и Хеллман (Hellman) [5] предложили принцип криптографии с открытым ключом в 1976 году, а исходная статья с описанием RSA, принадлежащая перу Ривеста (Rivest), Шамира (Shamir) и Адельмана (Adelman) [17], появилась два года спустя.

Более подробную информацию об официально одобренных ГПСЧ можно найти в приложении C к Федеральным стандартам обработки информации (Federal Information Processing Standards Publication 140-2) [6]. Об одной аппаратной реализации генератора случайных чисел на основе теплового шума можно прочесть в статье Тейлора (Taylor) и Кокса (Cox) [22].

9...Сжатие данных

В предыдущей главе мы рассмотрели, как преобразовать информацию, чтобы защитить ее от злоумышленника. Но защита информации — не единственная причина ее преобразования. Иногда этой причиной является желание ее улучшить, например изменить фотографию с помощью такого программного обеспечения, как Adobe Photoshop, чтобы убрать "красноглазие" или изменить тон кожи. Иногда добавляется избыточность информации, чтобы, если по каким-то причинам (например, при передаче по каналам связи) некоторые ее биты станут неверными, можно было обнаружить и исправить ошибки. В этой главе мы исследуем еще один способ преобразования информации — ее сжатие. Прежде чем приступить к изучению некоторых из методов, используемых для сжатия и распаковки информации, мы должны ответить на три вопроса.

1. *Зачем надо сжимать информацию?*

   Обычно мы сжимаем информацию для того, чтобы сохранить время и/или пространство (память).

   Время: при передаче информации по сети чем меньше битов передается, тем быстрее выполняется передача. Таким образом, отправитель часто сжимает данные перед отправкой и отправляет сжатые данные, а затем получатель распаковывает данные, которые он получает.

   Пространство: если имеющаяся память ограничивает количество информации, которое вы можете сохранить, его можно увеличить, храня информацию в сжатом виде. Например, в форматах MP3 и JPEG сжатие звука и изображения выполняется таким образом, что большинство людей не в состоянии заметить разницу (если таковая вообще существует) между первоначальными и сжатыми материалами.

2. *Каково качество сжатой информации?*

   Методы сжатия могут быть с потерями или без потерь. ***Сжатие без потерь*** позволяет после распаковки сжатых данных получить информацию, идентичную первоначальной. В случае ***сжатия с потерями*** распакованная информация отличается от оригинала, но в идеале — крайне незначительно. Сжатие MP3 и JPEG представляет собой сжатие с потерями, но метод сжатия, используемый программой zip, является сжатием без потерь.

   Вообще говоря, при сжатии текста требуется сжатие без потерь. Даже разница в один бит может быть значима. Например, следующие предложения различаются только одним битом в ASCII-кодах составляющих их букв[1]:

   ```
   Don't forget the pop.
   Don't forget the pot.
   ```

[1] *ASCII-коды букв* `p` *и* `t` *представляют собой 01110000 и 01110100 соответственно.*

3. *Почему информацию можно сжимать?*

   На этот вопрос легче ответить в случае сжатия с потерями: мы просто миримся со снижением точности передачи информации. Но что можно сказать о сжатии без потерь? Цифровая информация часто содержит избыточные или бесполезные биты. В ASCII, например, каждый символ занимает один 8-битовый байт, и все часто используемые символы имеют значение 0 в старшем (крайнем слева) бите[2], т.е. коды символов ASCII находятся в диапазоне от 0 до 255, но используемые для записи английского текста символы попадают в диапазон от 0 до 127. Поэтому зачастую старший бит в ASCII-тексте бесполезен, так что такой текст легко сжать на 12.5%.

   Еще более драматический пример использования избыточности при сжатии без потерь — передача черно-белых изображений, например, факсом. Факсы передают изображение как серии черных или белых точек, вместе образующих изображение. Многие факсы передают точки построчно сверху вниз. Если изображение состоит главным образом из текста, большая часть такого изображения белая, так что каждая строка, вероятно, содержит много участков из идущих подряд белых точек. Если строка содержит часть горизонтальной черной линии, в ней могут быть участки из идущих подряд черных точек. Вместо указания цвета каждой точки отдельно факсы сжимают информацию, указывая длину каждого одноцветного участка и его цвет. Например, в одном из стандартов факсов участок из 140 белых точек сжимается в одиннадцать битов `10010001000`.

Сжатие данных хорошо изучено, так что я могу коснуться здесь лишь небольшой его части. Я сосредоточусь на сжатии без потерь, но в разделе "Дальнейшее чтение" вы можете найти несколько ссылок, которые охватывают сжатие с потерями.

В этой главе, в отличие от предыдущих, нас не будет интересовать время работы алгоритмов. При необходимости я буду о нем упоминать, но в данном случае нас гораздо больше интересует размер сжатой информации, чем время ее сжатия и распаковки.

Коды Хаффмана

Давайте ненадолго вернемся к строкам, представляющим ДНК. Из главы 7, "Алгоритмы на строках", вы помните, что биологи представляют ДНК в виде строк из четырех символов `A`, `C`, `G` и `T`. Предположим, что ДНК представлена n символами, 45% которых являются `A`, 5% — `C`, 5% — `G` и 45% — `T`, но расположены эти символы в произвольном порядке. Если бы мы использовали для представления ДНК восьмибитовые ASCII-символы, нам потребовалось бы $8n$ битов. Конечно, можно поступить разумнее. Поскольку для представления ДНК нужны только четыре символа, нам хватит двух битов для представления каждого символа (`00`, `01`, `10`, `11`), а потому мы можем уменьшить представление до $2n$ бит.

---

[2] *Напомним, что кодировка ASCII включает только буквы английского алфавита, но не других алфавитов, например русского, как в случае кодировок CP-1251, CP-866 или KOI8-r. — Примеч. пер.*

Но если воспользоваться относительными частотами появления разных символов, можно добиться еще лучшего результата. Давайте кодировать символы следующими последовательностями битов: A = 0, C = 100, G = 101 и T = 11. Чаще встречающиеся символы кодируются более короткими последовательностями битов. Мы можем закодировать 20-символьную строку TAATTAGAAATTCTATTATA 33-битовой последовательностью 110011110101000111110011011110110 (вскоре вы поймете, почему я выбрал эту конкретную кодировку, и какими свойствами она обладает). С учетом частот появления наших четырех символов для кодирования n-символьной строки нам нужно $0.45 \cdot n \cdot 1 + 0.05 \cdot n \cdot 3 + 0.05 \cdot n \cdot 3 + 0.45 \cdot n \cdot 2 = 1.65n$ бит (обратите внимание, что в рассмотренном примере $33 = 1.65 \cdot 20$). Используя преимущества относительных частот появления символов, мы можем добиться результата, лучшего, чем $2n$ бит.

Использованный метод кодирования обладает не только тем свойством, что чем чаще встречается символ, тем короче представляющая его последовательность битов. Есть и другая интересная особенность: ни один код символа не является префиксом любого другого кода. Код A равен 0, и никакой другой код не начинается с 0; код для T — 11, и никакой другой код не начинается с 11, и т.д. Мы называем такой код ***префиксно-свободным кодом***[3].

Важное преимущество префиксно-свободных кодов проявляется в процессе распаковки. Поскольку код не является префиксом любого другого кода, можно однозначно сопоставить сжатые биты исходным символам при последовательной распаковке битов. В сжатой последовательности 110011110101000111110011011110110, например, ни один символ не имеет однобитовый код 1, и только код для T начинается с 11. Так что мы знаем, что первым символом несжатого текста должен быть T. Убирая 11, мы получаем 00111101010 0011111001101111 0110. С 0 начинается код одного символа — A, и поэтому первый символ оставшейся строки — A. После удаления бита 0, а затем битов 011110, соответствующих несжатой строке ATTA, у нас остаются биты 101000111110011011110110. Так как только код для G начинается с битов 101, очередным символом исходного текста должен быть G. И так далее.

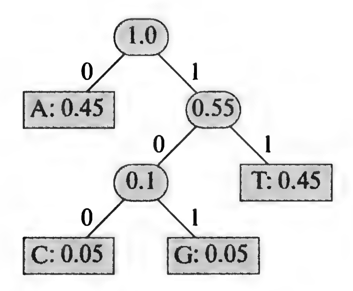
Если измерять эффективность методов сжатия средней длиной сжатой информации, то среди префиксно-свободных кодов код Хаффмана[4] является наилучшим. Единственный недостаток традиционного кода Хаффмана в том, что он требует предварительного знания частот всех символов, а потому часто требует два прохода по несжатому тексту: один — для определения частот символов и второй — для сопоставления символам их кодов. Чуть позже мы увидим, как можно избежать первого прохода ценой дополнительных вычислений.

После того как получена информация о частотах символов, метод Хаффмана строит бинарное дерево (если вы уже забыли, что это такое, вернитесь к с. 104 главы 6, "Кратчайшие пути"). Это дерево помогает строить коды и очень помогает при распаковке. Вот как выглядит дерево для нашего примера кодирования ДНК.

---

[3] *В CLRS такие коды называются просто "префиксными". Здесь я предпочитаю использовать более точный термин.*

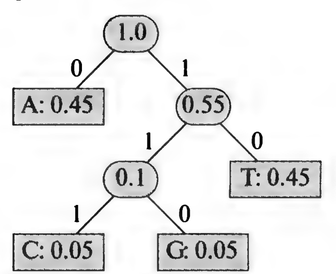
[4] *Назван так по имени его первооткрывателя Дэвида Хаффмана (David Huffman).*



Листья дерева, изображенные в виде прямоугольников, представляют символы (рядом с символом в каждом прямоугольнике показана его относительная частота). Внутренние узлы, не являющиеся листьями, изображены с закругленными углами, и в каждом внутреннем узле указана сумма частот в листьях ниже этого узла. Мы скоро увидим, зачем нам хранить эти частоты во внутренних узлах.

Рядом с каждым ребром дерева показано значение 0 или 1. Чтобы определить код символа, надо следовать по пути от корня до листа символа и объединять биты, встречающиеся на этом пути. Например, чтобы определить код для символа G, надо начать с корня и сначала идти вправо, по ребру, помеченному 1, и войти в правый дочерний узел корня. Затем из него надо идти влево по ребру, помеченному 0, в левый дочерний узел (внутренний узел с частотой 0.1), и наконец по ребру с меткой 1 мы попадаем в правый дочерний узел, который представляет собой лист символа G. Объединение меток ребер дает для символа G код 101.

Хотя лично я всегда помечаю ребра к левым дочерним узлам символом 0, а к правым — символом 1, значения меток не имеют большого значения. Можно было бы пометить ребра, например, таким образом.



С случае такого дерева мы получаем коды A = 0, C = 111, G = 110 и T = 10. Это по-прежнему префиксно-свободной код, и число битов в каждом коде символа остается прежним. Это связано с тем, что количество битов в коде символа равно глубине листа символа, иными словами, количеству ребер в пути от корня до листа. Однако будет проще, если мы будем всегда помечать ребра к левым дочерним узлам символом 0, а к правым — символом 1.

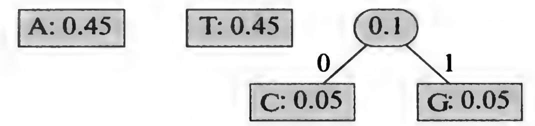
Когда нам известны частоты символов, мы можем начать строительство бинарного дерева снизу вверх. Мы начинаем с n листьев, соответствующих символам несжатого текста, как с n индивидуальных деревьев, так что изначально каждый лист является корневым узлом. Затем мы многократно ищем два корневых узла с наименьшими частотами и создаем новый корень с этими узлами в качестве дочерних; при этом новый корень получает

значение частоты, равное сумме частот своих дочерних узлов. Процесс продолжается до тех пор, пока все листья не окажутся под одним корнем. В процессе работы мы помечаем каждое ребро к левому дочернему узлу символом 0, а каждое ребро к правому дочернему узлу символом 1 (хотя после выбора двух корней с минимальными частотами не важно, какой из них будет левым дочерним узлом, а какой правым).

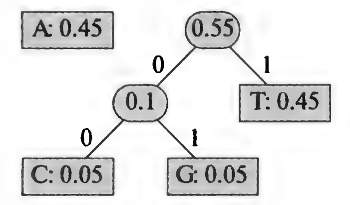
Распишем подробно процесс построения бинарного дерева для нашего примера с ДНК. Мы начинаем работу с четырех узлов, каждый из которых является листом, представляющим один символ.



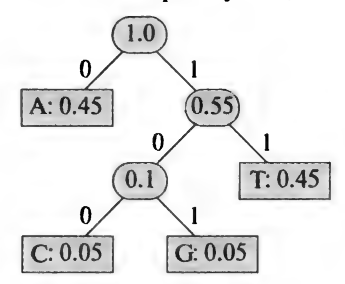
Узлы C и G имеют наименьшие частоты, так что мы создаем новый узел, делаем эти два узла его дочерними, и указываем частоту для нового узла, равную сумме частот исходных узлов.



Теперь у нас есть узел с минимальной частотой 0.1 и два узла с частотами 0.45. Мы можем выбрать любой из этих двух узлов для объединения с узлом с минимальной частотой; выберем узел символа T и сделаем его и узел с частотой 0.1 дочерними узлами нового корня с частотой 0.55 — суммой частот дочерних узлов.



Теперь у нас остались только два узла. Мы делаем их дочерними узлами вновь созданного, частота у которого равна сумме частот дочерних узлов, т.е. 1.0.



Теперь все листья находятся в одном дереве с новым корнем, так что искомое дерево построено.

Чтобы быть более точными, определим процедуру создания бинарного дерева. Процедура Build-Huffman-Tree принимает в качестве входных данных два n-элементных массива, *char* и *freq*, где $char[i]$ содержит i-й символ исходного алфавита, а $freq[i]$ — частота

его появления в исходном тексте. Процедура также получает значение n. Чтобы найти два корня с наименьшими частотами, процедура вызывает в качестве подпрограмм процедуры Insert и Extract-Min, работающие с очередью с приоритетами (см. с. 102).

Процедура Build-Huffman-Tree(*char*,*freq*,*n*)

Вход:

- *char*: n-элементный массив символов исходного алфавита.
- *freq*: n-элементный массив частот появления символов в исходном тексте.
- n: размер массивов *char* и *freq*.

Выход: корень бинарного дерева, построенного для кода Хаффмана.

1. Пусть Q — пустая очередь с приоритетами.
2. Для $i=1$ до n:
   A. Строим новый узел z, содержащий символ $char[i]$; частота узла делается равной $freq[i]$.
   B. Вызываем Insert(Q,z).
3. Для $i=1$ до $n-1$:
   A. Вызываем Extract-Min(Q) и присваиваем извлеченное вызовом значение переменной x.
   B. Вызываем Extract-Min(Q) и присваиваем извлеченное вызовом значение переменной y.
   C. Строим новый узел z, частота которого равна сумме частот x и y.
   D. Делаем x левым дочерним узлом z, а y — его правым дочерним узлом.
   E. Вызываем Insert(Q,z).
4. Вызываем Extract-Min(Q) и возвращаем извлеченный узел.

Когда процедура переходит к шагу 4, в очереди остается только один узел, который представляет собой корень всего построенного бинарного дерева.

Вы можете проследить, каким образом эта процедура строит бинарное дерево, рассмотренное выше. Корни, находящиеся в очереди с приоритетами в начале каждой итерации цикла на шаге 3, показаны в верхней части каждого рисунка.

Давайте бегло проанализируем время работы процедуры Build-Huffman-Tree. В предположении, что очередь с приоритетами реализована с помощью бинарной пирамиды, каждая операция вставки и извлечения минимального элемента занимает время $O(\lg n)$. Процедура вызывает каждую из этих операций $2n-1$ раз, так что на это тратится в общей сложности время $O(n\lg n)$. Все остальные действия выполняются в целом за время $\Theta(n)$, так что окончательно время работы процедуры Build-Huffman-Tree составляет $O(n\lg n)$.

Ранее я упоминал, что очень удобно иметь бинарное дерево, созданное процедурой Build-Huffman-Tree, при распаковке. Начиная с корня бинарного дерева, будем идти вниз по дереву согласно битам сжатой информации. Будем брать очередной бит и идти в левый дочерний узел, если этот бит нулевой, и в правый, если единичный. Добравшись до листа, помещаем соответствующий ему символ в выходной поток и повторяем поиск от корня. Возвращаясь к нашему примеру с ДНК, при разворачивании последовательности битов 11001111010100011111001101111011 мы извлекаем первый бит (1) и идем от корня вправо, после чего следующий извлеченный бит 1 вновь отправляет нас вправо, прямо в лист символа T. Мы выводим T и возобновляем поиск от корня. Очередной бит — 0, так что идем от корня влево и сразу попадаем в лист символа A, который и выводим, после чего опять возвращаемся в корень. Распаковка продолжается таким образом до тех пор, пока не будут обработаны все биты сжатой информации.

Если у нас есть уже построенное бинарное дерево, то для обработки каждого бита требуется константное время. Но как получить бинарное дерево для процесса распаковки? Одна из возможностей заключается в том, чтобы включить представление бинарного дерева в сжатую информацию. Другая возможность заключается в том, чтобы включить в сжатую информацию таблицу декодирования. Каждая запись такой таблицы включает символ, количество битов в его коде и сам код. Имея такую таблицу, можно построить бинарное дерево за время, линейно зависящее от общего количества битов во всех кодах.

Процедура Build-Huffman-Tree служит примером жадного алгоритма, в котором принимается решение, выглядящее наилучшим в данный момент. Поскольку мы хотим, чтобы редко встречающиеся символы находились от корня подальше, жадный подход всегда выбирает для нового узла два корня с наиболее низкой частотой. Алгоритм Дейкстры (с. 103) является еще одним жадным алгоритмом, потому что всегда ослабляет ребра, выходящие из вершины с наименьшим значением *shortest* из остающихся в этот момент в очереди с приоритетами.

Я реализовал алгоритм кодирования Хаффмана и применил его к онлайн-версии "*Моби Дик*". Исходный текст имел размер 1 193 826 байтов, а сжатая версия — только 673 579 байтов, или 56.42% от размера оригинала, не включая саму таблицу кодов. Другими словами, при сжатии в среднем каждый символ можно представить с помощью только 4.51 бита. Как и ожидалось, наиболее часто встречался пробел (15.96%), затем — символ e (9.56%). Наименее частыми символами (каждый из них встретился только по 2 раза) были $, &, [и].

Адаптивные коды Хаффмана

Практики часто обнаруживают, что выполнять два прохода по входным данным (один — для вычисления частот появления символов, а второй — для их кодирования) — слишком медленное решение. Вместо этого программы сжатия и распаковки могут работать адаптивно, обновляя частоты символов и бинарное дерево в процессе сжатия или распаковки за один проход.

Программа сжатия начинает работу с пустого бинарного дерева. Каждый символ, который она считывает из входных данных, является либо новым, либо уже присутствующим в бинарном дереве. Если символ уже находится в дереве, то программа сжатия выдает его

код в соответствии с текущим состоянием бинарного дерева, увеличивает частоту символа и при необходимости обновляет бинарное дерево в соответствии с новой частотой. Если символ в бинарном дереве отсутствует, программа сжатия вставляет символ в выходной поток как есть, в *не*закодированном виде, затем добавляет этот символ в бинарное дерево и соответствующим образом обновляет последнее.

Программа распаковки является зеркальным отображением программы сжатия. Она также поддерживает бинарное дерево в процессе обработки сжатой информации. Когда программа встречает биты символа из бинарного дерева, она идет по дереву в соответствии с этими битами, находит, какому символу они соответствуют, выводит его, увеличивает частоту символа и обновляет бинарное дерево. Если же программа распаковки встречает символ, которого пока что нет в дереве, она просто выводит этот символ, а затем добавляет его в бинарное дерево и обновляет последнее.

Но здесь есть один неприятный момент. Биты есть биты, представляют ли они символы ASCII или биты кода Хаффмана. Как же программа распаковки может определить, являются ли просматриваемые ею биты закодированный или незакодированной символ? Например, последовательность битов 101 — представляет ли она символ, в настоящее время кодируемый как 101, или это первые три бита незакодированного символа? Решение заключается в использовании предшествующего каждому незакодированному символу ***управляющего кода***, который представляет собой специальный код, указывающий, что следующий за ним набор битов является незакодированным символом. Если исходный текст содержит *k* различных символов, то в сжатой информации встретятся только *k* управляющих символов, каждый из которых предваряет первое вхождение символа. Такие коды обычно появляются очень редко, так что они не должны быть слишком короткими за счет более часто встречающихся символов. Хороший способ обеспечить длинные управляющие коды — включить их в бинарное дерево, но с фиксированной нулевой частотой. Тогда при любом обновлении бинарного дерева хотя управляющий код и будет изменяться, его лист всегда будет располагаться далеко от корня.

Факсимильные аппараты

Ранее я уже упоминал, что факсы сжимают информацию, указывая длину и цвет последовательности идентичных точек в строке. Эта схема сжатия известна как ***кодирование длин серий*** (run-length encoding, RLE). Факсимильные аппараты комбинируют кодирование длин серий с кодами Хаффмана. В стандарте для факсимильных аппаратов для обычных телефонных линий 104 кода указывают серии разной длины из белых точек, и 104 кода — серии разной длины из черных точек. Коды для белых точек префиксно-свободные, как и коды для черных точек, хотя при этом некоторые коды для белых точек являются префиксами кодов для черных точек и наоборот.

Чтобы определить, какие коды использованы для кодирования последовательностей, комитет по стандартизации взял множество из восьми представительных документов и подсчитал, как часто встречается каждая из последовательностей. Затем для этих последовательностей были построены коды Хаффмана. Наиболее частыми (а следовательно,

получившими наиболее короткие коды) оказались последовательности из двух, трех и четырех черных точек (с кодами 11, 10 и 011 соответственно). Прочими часто встречающимися последовательностями оказались одна черная точка (код 010), пять и шесть черных точек (коды 0011 и 0010), от двух до семи белых точек (коды этих последовательностей состоят из четырех битов каждый) и другие относительно короткие последовательности. Довольно часто встречается последовательность из 1664 белых точек, представляющая пустую строку. Прочие короткие коды принадлежат последовательностям белых точек, длины которых являются степенями 2 или суммами двух степеней 2 (например, $192 = 2^7 + 2^6$). Последовательности могут кодироваться как конкатенации кодов более коротких последовательностей. Ранее в качестве примера кода для последовательности 140 белых точек я привел код 10010001000. На самом деле это конкатенация кодов для последовательности из 128 белых точек (10010) и последовательности из 12 белых точек (001000).

В дополнение к сжатию информации только в пределах каждой строки изображения некоторые факсимильные аппараты выполняют сжатие изображения в обоих измерениях. Последовательности точек одного цвета могут располагаться как в вертикальном, так и в горизонтальном направлениях, так что вместо того, чтобы рассматривать каждую строку изолированно, строка кодируется в соответствии с тем, где она отличается от предыдущей. В большинстве случаев строка отличается от предыдущей всего несколькими точками. Такая схема влечет за собой риск распространения ошибок: ошибка кодирования или передачи делает неверными несколько последовательных строк. По этой причине факсы, которые используют эту схему и используют передачу по телефонным линиям, ограничивают количество зависимых последовательных строк: после определенного количества зависимых строк очередная строка изображения передается полностью, закодированной с помощью схемы кодирования Хаффмана.

LZW-сжатие

Другой подход к сжатию без потерь, особенно для текста, использует информацию, которая повторяется в тексте, хотя и необязательно в последовательных местах. Рассмотрим, например, знаменитую цитату из инаугурационной речи Джона Кеннеди (John F. Kennedy).

Ask not what your country can do for you — ask what you can do for your country.[5]

За исключением слова *not*, каждое слово в цитате повторяется дважды. Предположим, мы создали таблицу слов.

Индекс	Слово
1	ask
2	not
3	what
4	your
5	country

[5] *Не спрашивай, что твоя страна может сделать для тебя; спроси, что ты можешь сделать для своей страны. — Примеч. пер.*

6	can
7	do
8	for
9	you

Тогда мы можем закодировать цитату (игнорируя прописные буквы и знаки препинания) как

1 2 3 4 5 6 7 8 9 1 3 9 6 7 8 4 5

Поскольку эта цитата состоит из небольшого количества слов, а байт может содержать целые числа от 0 до 255, мы можем хранить каждый индекс в одном байте. Таким образом, всю цитату можно хранить только в 17 байтах, по байту на слово, плюс память, необходимая для хранения таблицы. Если использовать по байту на символ исходной цитаты, без знаков препинания, но с пробелами между словами, нам потребуется 77 байт.

Конечно, память, требующаяся для хранения таблицы, имеет значение, так как в противном случае мы могли бы просто перенумеровать все возможные слова и сжимать файл, сохраняя только индексы слов. Но для некоторых слов эта схема приводит не к сжатию, а к расширению. Почему? Давайте предположим, что всего слов имеется меньше, чем 2^{32}, так что мы можем хранить каждый индекс как одно 32-битовое слово. Итак, каждое слово при сжатии заменяется четырьмя байтами, а потому схема не работает в случае слов из трех букв или более коротких.

Препятствием для нумерации всех возможных слов является то, что реальный текст может включать "слова", которые не являются словами, или, по крайней мере, не являются словами английского (или иного) языка. В качестве забавного примера можно привести, например, первое четверостишие стихотворения Льюиса Кэрролла (Lewis Carroll) "Jabberwocky":

Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe.[6]

Можно вспомнить также компьютерные программы, которые часто используют имена переменных, не являющиеся английскими словами. Добавив прописные и строчные буквы, символы пунктуации и очень длинные географические названия[7], увидим, что, если сжимать текст с помощью нумерации всех слов, нам потребуется очень большое количество индексов. Конечно, это число куда больше 2^{32} — оно просто неограниченное, поскольку в принципе в тексте *может* встретиться любое сочетание символов.

[6] *Стихотворение "Бармаглот":*
Варкалось. Хливкие шорьки
Пырялись по наве,
И хрюкотали зелюки,
Как мюмзики в мове.

(Перевод Д. Орловской)

[7] *Типа названия деревни Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch в Уэльсе.*

Однако еще не все потеряно, поскольку мы все же можем воспользоваться повторяющейся информацией. Нам просто нужно не так зацикливаться на повторяющихся словах. Может помочь любая повторяющаяся последовательность символов. Несколько схем сжатия информации основаны на повторяющихся последовательностях символов. Та, которая будет рассмотрена нами, известна как ***LZW***[8] и является основой для многих программ сжатия, используемых на практике.

Метод LZW выполняет один проход по входным данным для сжатия и распаковки. В обоих случаях он строит словарь последовательностей просмотренных символов и использует для представления последовательностей символов индексы в этом словаре. Рассматривайте словарь как массив символьных строк. Мы можем индексировать этот массив и говорить о его i-й записи. Ближе к началу входных данных последовательности, как правило, короткие, и их представление с помощью индексов может привести к расширению, а не сжатию. Но по мере работы LZW с входными данными последовательности в словаре становятся все длиннее, и их представление с помощью индексов обеспечивает существенное сжатие информации. Например, я пропустил текст "*Моби Дик*" через программу сжатия LZW, и она 20 раз сгенерировал в качестве вывода индекс, представляющий 10-символьную последовательность `_from_the_` (символ `_` указывает пробел). Она также 33 раза вывела индекс, представляющий восемь символов последовательности `_of_the_`.

Как программа сжатия, так и программа распаковки заполняют словарь односимвольными последовательностями для каждого символа из используемого набора символов. Так, при использовании полного набора символов ASCII словарь начинается с 256 односимвольных последовательностей; i-я запись в словаре содержит символ, ASCII-код которого представляет собой i.

Перед тем как перейти к общему описанию работы программы сжатия, давайте взглянем на пару возможных ситуаций. Программа сжатия создает строки, вставляет их в словарь и в качестве вывода возвращает индексы в словаре. Предположим, что программа сжатия начинает построение строки с символа `T`, считанного из входного потока. Поскольку словарь содержит все односимвольные последовательности, программа сжатия находит `T` в словаре. Всякий раз, когда программа находит строку в словаре, она считывает следующий символ из входных данных и добавляет этот символ к построенной строке. Предположим теперь, что следующий входной символ — `A`. Программа добавляет `A` к строящейся строке и получает строку `TA`. Предположим, что эта строка также имеется в словаре. Затем программа считывает очередной входной символ, скажем, `G`. Добавление `G` к строящейся строке дает строку `TAG`, и на этот раз предположим, что этой строки в словаре *нет*. Программа делает три вещи: (1) выводит индекс строки `TA` в словаре; (2) вставляет строку `TAG` в словарь; и (3) начинает построение новой строки, первоначально содержащей только один символ (`G`), который привел к отсутствию строки в словаре.

---

[8] *Как обычно, название увековечивает создателей. Терри Уэлч (Terry Welch) создал LZW на основе схемы сжатия LZ78, предложенной Абрамом Лемпелем (Abraham Lempel) и Якобом Зивом (Jacob Ziv).*

Вот как программа сжатия работает в общем случае. Она генерирует последовательность индексов в словарь. Конкатенация строк с этими индексами дает исходный несжатый текст. Программа строит строки в словаре по одному символу за раз, так что всякий раз, когда она вставляет строку в словарь, эта строка такая же, как уже имеющаяся в словаре, но продленная на один символ. Программа сжатия управляет строкой *s* из последовательных символов входных данных, поддерживая инвариант, заключающийся в том, что словарь всегда содержит *s* в некоторой из своих записей. Даже если *s* состоит из единственного символа, он имеется в словаре, поскольку словарь изначально заполнен односимвольными последовательностями для каждого символа из используемого набора символов. Первоначально *s* представляет собой первый символ входного потока. При чтении нового символа *c* программа сжатия проверяет, имеется ли в настоящее время в словаре строка *sc*, образованная путем добавления *c* к концу *s*. Если есть, то программа добавляет *c* в конец *s* и называет результат *s*; другими словами, она присваивает *s* значение *sc*. Программа сжатия строит все более длинную строку, которая в конечном итоге будет вставлена в словарь. В противном случае (*s* присутствует в словаре, но *sc* — нет) программа сжатия выводит индекс *s* в словаре, вставляет *sc* в свободную запись словаря и устанавливает *s* равной одному входному символу *c*. Вставляя *sc* в словарь, программа добавляет в него строку, которая расширяет *s* на один символ, а устанавливая строку *s* равной *c*, перезапускает процесс построения строки для поиска в словаре. Поскольку *c* представляет собой односимвольную строку, имеющуюся в словаре, программа поддерживает инвариант, заключающийся в том, что *s* имеется в словаре. После того как входной поток исчерпан, программа выводит индекс оставшейся строки *s*.

Процедура LZW-COMPRESSOR(*text*)

Вход: *text* — последовательность символов из набора ASCII.

Выход: последовательность индексов в словаре.

1. Для каждого символа *c* из набора символов ASCII:

   A. Вставить символ *c* в словарь с индексом, равным числовому коду *c* в наборе символов ASCII.

2. Установить *s* равным первому символу *text*.
3. Пока *text* не исчерпан, выполнять следующие действия.

   A. Взять из *text* очередной символ и присвоить его переменной *c*.

   B. Если *sc* имеется в словаре, установить $s = sc$.

   C. В противном случае (*sc* пока еще нет в словаре) выполнить следующие действия.

      i. Вывести индекс *s* в словаре.

      ii. Вставить *sc* в очередное свободное место в словаре.

      iii. Присвоить *s* строку, состоящую из одного символа *c*.

4. Вывести индекс строки *s* в словаре.

Давайте рассмотрим конкретный пример — скажем, сжатие текста TATAGATCTTAATATA (здесь мы встретимся с последовательностью TAG, которую уже видели выше). В приведенной далее таблице показано, что происходит после каждой итерации цикла на шаге 3. Показаны значения строки *s* в начале итераций.

Итерация	*s*	*c*	Выход	Новая строка словаря
1	T	A	84 (T)	256: TA
2	A	T	65 (A)	257: AT
3	T	A		
4	TA	G	256 (TA)	258: TAG
5	G	A	71 (G)	259: GA
6	A	T		
7	AT	C	257 (AT)	260: ATC
8	C	T	67 (C)	261: CT
9	T	T	84 (T)	262: TT
10	T	A		
11	TA	A	256 (TA)	263: TAA
12	A	T		
13	AT	A	257 (AT)	264: ATA
14	A	T		
15	AT	A		
Шаг 4	ATA		264 (ATA)	

После шага 1 словарь содержит односимвольные строки для каждого из 256 символов ASCII в записях с номерами от 0 до 255. Шаг 2 делает строку *s* состоящей из единственного входного символа T — первого входного символа. На первой итерации основного цикла на шаге 3 *c* становится равным очередному входному символу — A. Конкатенация *sc* представляет собой строку TA, которой пока нет в словаре, так что выполняется шаг 3C. Поскольку строка *s* содержит только T, а ASCII-код T равен 84, шаг 3Ci выводит индекс 84. Шаг 3Cii вставляет строку TA в очередную свободную запись в словаре (индекс которой — 256), а шаг 3Ciii начинает построение строки *s* заново, устанавливая ее значение просто равным символу A. Во второй итерации цикла на шаге 3 *c* представляет собой следующий входной символ — T. Строка $sc =$ AT отсутствует в словаре, и поэтому шаг 3C выводит индекс 65 (код ASCII для A), вносит строку AT в запись 257 и устанавливает *s* равной односимвольной строке T.

Преимущество применения словаря мы видим во время двух следующих итераций цикла на шаге 3. В третьей итерации *c* присваивается следующий входной символ A. Теперь строка $sc =$ TA присутствует в словаре, так что процедура ничего не выводит. Вместо этого на шаге 3B к концу строки *s* добавляется считанный символ, и строка *s* принимает

вид TA. В четвертой итерации символ c становится равным G. Строки sc = TAG в словаре нет, а потому шаг 3Ci выводит индекс строки s в словаре, равный 256. Теперь одно число на выходе указывает не на один, а на два символа — TA.

За время работы процедуры LZW-COMPRESSOR некоторые индексы могут не быть выведены в выходной поток ни разу, а некоторые — несколько раз. Если объединить все символы в скобках в столбце "Выход" приведенной выше таблицы, мы получим исходный текст TATAGATCTTAATATA.

Этот небольшой пример слишком мал, чтобы показать реальную эффективность сжатия LZW. На вход подается 16 байт, а выход состоит из 10 индексов словаря. Каждый индекс при этом требует более одного байта. Даже если мы используем на выходе два байта на один индекс, то в результате получим 20 байт. Если же каждый индекс занимает четыре байта, то общий размер "сжатой" информации равен 40 байтам.

Более длинные тексты, как правило, дают лучшие результаты. LZW-сжатие уменьшает размер "*Моби Дика*" с 1 193 826 до 919 012 байт. При этом в словаре 230 007 записей, так что индексы должны иметь размер по меньшей мере четыре байта[9]. Вывод состоит из 229 753 индексов, или 919 012 байт. Этот результат уступает сжатию с помощью кодирования Хаффмана (673 579 байт), но чуть позже мы увидим некоторые идеи, каким образом можно повысить степень сжатия.

LZW-сжатие имеет смысл только в том случае, если мы можем распаковать сжатую информацию. К счастью, хранить для этого словарь вместе со сжатой информацией не требуется. (Если бы это требовалось, то сжатая информация вместе со словарем по размеру не превышали бы исходный текст только в очень редких случаях.) Как упоминалось ранее, распаковка LZW в состоянии построить словарь непосредственно из сжатой информации.

Вот как работает LZW-распаковка. Подобно программе сжатия, программа распаковки начинает со словаря, состоящего из 256 односимвольных последовательностей, соответствующих символам из набора символов ASCII. Она получает из входного потока последовательность индексов в словаре и зеркально отражает действия программы сжатия по построению словаря. Всякий раз, когда распаковщик выполняет вывод, выводится строка, добавленная им к словарю.

Чаще всего считанный из входного потока индекс указывает на строку, уже имеющуюся в словаре (вскоре мы увидим, что происходит в противном случае), так что распаковщик находит строку в словаре, соответствующую этому индексу, и выводит ее. Но как распаковщик может построить словарь? Давайте на минутку задумаемся о том, как работает программа сжатия. К моменту, когда программа выводит индекс на шаге 3C, она обнаружила, что, хотя строка s в словаре имеется, строки sc в нем нет. Программа выводит индекс строки s в словаре, вставляет sc в словарь и начинает построение новой строки для сохранения, начиная с символа c. Распаковщик должен работать соответствующим образом. Для каждого индекса, полученного из входного потока, он выводит строку s, находящуюся в словаре в записи с этим индексом. Но он также знает, что в момент вывода программой

---

[9] *Я предполагаю, что целые числа имеют стандартное представление в виде одного, двух, четырех или восьми байт. Теоретически числа до 230 007 можно представить всего тремя байтами, и в таком случае общий размер сжатой информации будет равен 689 259 байт.*

упаковки индекса для *s* строки *sc* в словаре нет (здесь *c* — символ, следующий сразу после *s*). Распаковщик знает, что программа сжатия вставила строку *sc* в словарь, так что в конечном итоге распаковщик должен поступить так же. Он пока что не в состоянии вставить строку *sc*, потому что не видел символ *c*. Этот символ — первый символ очередной выводимой распаковщиком строки. Однако пока что следующая строка ему неизвестна. Следовательно, распаковщик должен отслеживать две последовательно выводимые строки. Если программа распаковки выводит строки *X* и *Y* в указанном порядке, то она должна добавить первый символ *Y* к *X*, а затем вставить получившуюся строку в словарь.

Давайте рассмотрим конкретный пример, для чего обратимся к таблице на с. 169, которая показывает, как программа сжатия работает со входной строкой TATAGATCTTAATATA. В итерации 11 программа сжатия выводит для строки TA индекс 256 и вставляет в словарь строку TAA. Это связано с тем, что в указанный момент в словаре уже имелась строка $s = \mathtt{TA}$, но не было строки $sc = \mathtt{TAA}$. Этот последний символ A является началом очередной строки (AT), которую программа сжатия выведет как индекс 257 в итерации 13. Поэтому, когда программа распаковки встречает индексы 256 и 257, она должна вывести строку TA, а также запомнить ее, чтобы, когда следующей будет выводиться строка AT, символ A из нее мог быть добавлен к строке TA и получившаяся в результате конкатенации строка TAA могла быть добавлена в словарь.

В редких случаях очередной индекс, поступающий в распаковщик из входного потока, еще не имеет соответствующей записи в словаре. Эта ситуация возникает настолько редко, что при распаковке "*Моби Дика*" она произошла всего для 15 из 229 753 индексов. Такое происходит, когда индекс, выводимый программой сжатия, соответствует последней вставленной в словарь строке. Эта ситуация возможна только тогда, когда строка, соответствующая индексу, начинается и заканчивается одним и тем же символом. Почему? Вспомним, что программа сжатия выводит индекс для строки *s* только тогда, когда *s* находится в словаре, но *sc* в нем отсутствует. Затем она вставляет строку *sc* в словарь, скажем, с индексом *i* и заново строит новую строку *s*, начинающуюся с *c*. Если следующий индекс, выводимый программой сжатия, представляет собой *i*, то строка, соответствующая этому индексу в словаре, должна начинаться с *c*, и при этом мы только что видели, что эта строка представляет собой *sc*. Так что если очередной индекс словаря во входном потоке распаковщика соответствует записи, которой еще нет в словаре, распаковщик может вывести строку, вставленную последней, добавить к ней ее же первый символ и вставить получившуюся строку в словарь.

Поскольку эти ситуации очень редки, приводимый пример несколько надуманный. Итак, пусть мы имеем строку TATATAT. Программа сжатия делает следующее: выводит индекс 84 (T) и вставляет TA в запись с индексом 256; выводит индекс 65 (A) и вставляет строку AT в запись с индексом 257; выводит индекс 256 (TA) и вставляет TAT в запись с индексом 258; и наконец выводит индекс 258 (только что вставленная строка TAT). Распаковщик, считывая индекс 258, берет только что выведенную строку TA, добавляет к ней первый символ этой же строки T, выводит получившуюся строку TAT и вставляет ее в словарь.

Хотя эта редкая ситуация возникает только тогда, когда строка начинается и заканчивается одним и тем же символом, обратное — что она возникает всякий раз, когда строка начинается и заканчивается одним и тем же символом — неверно. Например, при сжатии "*Моби Дика*" строки, начинавшиеся и заканчивавшиеся одним и тем же символом, для которых в выходной поток выводился соответствующий индекс, встретились 11 376 раз (около 5% от общего количества); при этом они не являлись последними вставленными в словарь строками.

Процедура LZW-DECOMPRESSOR(*indices*)

Вход: *indices*: последовательность индексов в словаре, созданном процедурой LZW-COMPRESSOR.

Выход: исходный текст, переданный на вход процедуры LZW-COMPRESSOR.

1. Для каждого символа *c* из набора символов ASCII:
   A. Вставить символ *c* в словарь с индексом, равным числовому коду *c* в наборе символов ASCII.
2. Установить значение переменной *current* равным первому индексу в *indices*.
3. Вывести строку из словаря, соответствующую индексу *current*.
4. Пока последовательность *indices* не исчерпана, выполнять следующие действия.
   A. Установить значение переменной *previous* равным значению *current*.
   B. Получить очередное число из последовательности *indices* и присвоить его переменной *current*.
   C. Если словарь содержит запись с индексом *current*, выполнить следующие действия.
      i. Установить *s* равной строке в словаре с индексом *current*.
      ii. Вывести строку *s*.
      iii. Вставить в очередную свободную запись словаря строку, индексированную значением *previous*, к которой добавлен первый символ *s*.
   D. В противном случае (словарь не содержит запись с индексом *current*) выполнить следующие действия.
      i. Установить *s* равной строке в словаре с индексом *previous*, к которой добавлен первый символ этой же записи в словаре.
      ii. Вывести строку *s*.
      iii. Вставить в очередную свободную запись словаря строку *s*.

В приведенной далее таблице показано, что происходит в каждой итерации цикла на шаге 4 в случае, когда входными данными для распаковки служат индексы в столбце "Выход" на с. 169. Строки, индексированные в словаре значениями переменных *previous* и *current*, выводятся в последовательных итерациях, а значения переменных *previous* и *current* приведены для каждой итерации после шага 4B.

Итерация	*previous*	*current*	Выход (s)	Новая строка словаря
Шаги 2,3		84	T	
1	84	65	A	256: TA
2	65	256	TA	257: AT
3	256	71	G	258: TAG
4	71	257	AT	259: GA
5	257	67	C	260: ATC
6	67	84	T	261: CT
7	84	256	TA	262: TT
8	256	257	AT	263: TAA
9	257	264	ATA	264: ATA

За исключением последней итерации входной индекс уже присутствует в словаре, так что шаг 4D выполняется только в последней итерации. Обратите внимание, что словарь, построенный процедурой LZW-Decompressor, совпадает со словарем, построенным процедурой LZW-Compressor.

Я не буду описывать, как искать информацию в словаре в процедурах LZW-Compressor и LZW-Decompressor. В последнем случае это особенно легко: надо просто отслеживать последний использованный индекс в словаре, и если индекс в *current* не превышает последний использованный индекс, то искомая строка имеется в словаре. Что касается процедуры LZW-Compressor, то перед ней стоит более сложная задача: для заданной строки определить, имеется ли она в словаре, и если да, то каков ее индекс. Конечно, можно выполнить простой линейный поиск в словаре, но если там содержится n элементов, каждый такой поиск будет требовать время $O(n)$. Эффективнее воспользоваться какой-либо из специализированных структур данных. Одной такой структурой может быть ***луч*** (trie), который напоминает бинарное дерево, которое мы строили для кодирования Хаффмана, с тем отличием, что каждый узел может иметь много дочерних узлов, а каждое ребро помечено символом ASCII. Еще одной эффективной структурой является ***хеш-таблица***, которая обеспечивает простой и очень эффективный в среднем способ поиска строки в каталоге.

Усовершенствование метода LZW

Как я уже говорил, меня не слишком впечатлило применение метода LZW для сжатия текста "*Моби Дика*". Частично эта проблема связана с большим размером словаря. При наличии 230 007 записей каждый индекс требует по крайней мере четырех байтов, так что выходной поток из 229 753 индексов сжимает исходный текст до 919 012 байт. Далее, можно заметить некоторые свойства индексов, генерируемых процедурой LZW-Compressor. Во-первых, многие из них представляют собой небольшие величины, а это означает, что в 32-битовом представлении бóльшая часть их битов — нулевые. Во-вторых, одни индексы будут использоваться гораздо чаще других.

Если выполняются оба эти свойства, к хорошим результатам может привести кодирование Хаффмана. Я модифицировал программу кодирования методом Хаффмана так, чтобы она работала с четырехбайтовыми целыми числами, а не символами, и применил ее к сжатому с помощью LZW-метода тексту "*Моби Дика*". Получившийся в результате файл занимает только 460 971 байт, или 38.61% от размера исходного текста (1 193 826 байт), что лучше, чем при использовании одного лишь метода кодирования Хаффмана. Конечно, такой метод сжатия предполагает наличие двух этапов — сжатие исходного текста с помощью LZW и затем сжатие полученной последовательности индексов методом Хаффмана. Соответственно, распаковка также будет представлять собой двухэтапный процесс: сначала сжатая информация распаковывается с помощью кодирования Хаффмана, а затем выполняется распаковка LZW.

Другие подходы к LZW-сжатию стремятся уменьшить количество битов, необходимых для хранения индексов, выводимых программой сжатия. Поскольку многие из индексов представляют собой небольшие числа, один из подходов заключается в использовании меньшего количества битов для меньших чисел, резервируя, скажем, первые два бита для указания количества использованных для представления числа битов. Вот одна такая схема.

- Если первые два бита — 00, то индекс находится в диапазоне от 0 до 63 ($2^6 - 1$), требуя для своего представления шести битов (т.е. в сумме — один байт).
- Если первые два бита — 01, то индекс находится в диапазоне от 64 (2^6) до 16 383 ($2^{14} - 1$), требуя для своего представления 14 бит (т.е. в сумме — два байта).
- Если первые два бита — 10, то индекс находится в диапазоне от 16 384 (2^{14}) до 4 194 303 ($2^{22} - 1$), требуя для своего представления 22 бит (т.е. в сумме — три байта).
- Наконец, если первые два бита — 11, то индекс находится в диапазоне от 4 194 304 (2^{22}) до 1 073 741 823 ($2^{30} - 1$), требуя для своего представления 30 бит (т.е. в сумме — четыре байта).

В двух других подходах индексы на выходе программы сжатия имеют один и тот же размер в силу ограничения размера словаря. В одном варианте, когда словарь достигает максимального размера, новые записи в него больше не вставляются. В другом варианте после того, как словарь достигает максимального размера, он очищается (за исключением первых 256 записей), после чего процесс заполнения словаря начинается заново с той точки текста, где словарь оказывается заполненным. Во всех случаях программа распаковки должна зеркально отражать действия, выполняемые программой сжатия.

Дальнейшее чтение

Книга Саломона (Salomon) [18] при охвате широкого спектра методов сжатия выделяется своей ясностью и краткостью. Книга Шторера (Storer) [21], опубликованная за 20 лет до книги Саломона, представляет собой классический труд в данной области алгоритмов. В разделе 16.3 CLRS [4] описаны коды Хаффмана, хотя и без доказательства того факта, что они являются наилучшими среди префиксно-свободных кодов.

10...Трудная? Задача...

Когда я покупаю что-то через Интернет, продавец должен доставить купленное мне домой. В большинстве случаев продавец пользуется услугами компании, специализирующейся на доставке товаров. Не буду говорить, какая именно компания чаще всего доставляет мне купленное, скажу только, что перед моим домом очень часто можно увидеть очередной коричневый грузовик.

Коричневые грузовики

Только в США компания по доставке товаров оперирует более чем 91 000 таких коричневых грузовиков. По крайней мере пять дней в неделю каждый грузовик начинает и заканчивает свой дневной маршрут в одном из гаражей и доставляет товары во множество мест. Естественно, что компания весьма заинтересована минимизировать затраты, связанные с грузовиками, делающими множество остановок каждый день. Например, один источник, с которым я консультировался, утверждал, что после того, как компания наметила новые маршруты для своих водителей, позволяющие уменьшить количество левых поворотов, она сократила общий пробег своих транспортных средств на 464 000 миль за 18-месячный период, сэкономив более 51 000 галлонов топлива, с дополнительным преимуществом, заключающимся в уменьшении выбросов углекислого газа на 506 тонн.

Но как компания может минимизировать ежедневную стоимость движения каждого грузовика? Предположим, что некоторый грузовик в некоторый день должен доставить товары в n мест. Добавление гаража в качестве одного из мест дает $n+1$ точек, которые должны быть посещены грузовиком. Для каждого из этих $n+1$ мест компания может рассчитать расходы на поездку грузовика из каждого из прочих n мест, так что компания имеет таблицу расходов по проезду грузовика из одного места в другое размером $(n+1)\times(n+1)$; диагональные записи в этой таблице смысла не имеют, так как i-я строка и i-й столбец соответствуют одному и тому же месту. Компания хочет определить маршрут, который начинается и заканчивается в гараже и посещает все n мест ровно один раз, такой, что общая стоимость всего маршрута является минимальной.

Можно написать компьютерную программу, которая позволит решить эту задачу. В конце концов, если мы рассмотрим конкретный маршрут с известным порядком остановок, то надо просто просуммировать соответствующие маршруту элементы таблицы расходов. Так что можно просто перечислить все возможные маршруты и найти среди них тот, который имеет минимальную стоимость. Количество возможных маршрутов конечно, так что в какой-то момент программа завершит работу и выдаст ответ. И похоже, такую программу не так уж и трудно написать, не правда ли?

Да, ее действительно нетрудно написать.

Трудно дождаться, когда она завершит работу...

Проблема в наличии огромного количества возможных маршрутов, которые посещают n мест: их $n!$ (n факториал). Почему? Грузовик выезжает из гаража. Перед ним выбор

из n мест для первой остановки. После первой остановки он может выбрать любое из оставшихся $n-1$ мест для второй остановки, так что имеется $n \cdot (n-1)$ возможных комбинаций первых двух остановок в определенном порядке. Как только мы сделаем эти две остановки, у нас останется выбор из $n-2$ мест для третьей остановки, а общее количество маршрутов из трех остановок составит $n \cdot (n-1) \cdot (n-2)$. Продолжая в том же духе, мы находим, что общее количество возможных маршрутов по всем n местам равно $n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 = n!$.

Вспомним, что $n!$ растет быстрее, чем даже экспоненциальная функция. В главе 8, "Основы криптографии", я писал, что 10! равно 3628800. Для компьютера это не так уж много. Но коричневые грузовики развозят гораздо больше 10 товаров в день. Предположим, что грузовик доставляет товары по 20 адресам в день (в США в среднем в грузовике размещается около 170 товаров, так что даже с учетом того, что в одно место могут доставляться несколько товаров одновременно, 20 адресов в день не кажутся завышенной оценкой). В этом случае программа должна обработать 20! возможных маршрутов, а 20! — это 2432902008176640000. Если компьютеры компании в состоянии обрабатывать триллион маршрутов в секунду, то потребуется всего лишь 28 дней на поиск среди них оптимального. И это только для одного из более чем 91000 грузовиков!

Так что при использовании такого подхода компании для приобретения и эксплуатации вычислительной мощности, необходимой для поиска маршрутов для всех грузовиков на каждый день, потребуются такие расходы, которые не покроешь никакой выгодой от более эффективных маршрутов. Нет, эта идея — перечисление всех возможных маршрутов и выбор наилучших — хотя и звучит математически разумно, совершенно непрактична. Нет ли лучшего способа найти для каждого грузовика маршрут наименьшей стоимости?

Никто не знает (а если и знает, то не говорит). Никто не нашел способ получше, но никто и не доказал, что такого способа не существует. Это вас не сильно разочаровало?

На самом деле это разочарование гораздо большее, чем вы можете себе представить. Задача поиска маршрутов с минимальной стоимостью более известна как ***задача коммивояжера*** (она называется так потому, что в ее первоначальной формулировке коммивояжер должен был посетить n городов, начиная и заканчивая путь в одном и том же городе, и при этом его маршрут должен быть кратчайшим возможным). Пока что не найден ни один алгоритм ее решения за время $O(n^c)$ для какой бы то ни было константы c. Мы не знаем алгоритма, который бы находил наилучший возможный маршрут по n городам ни за время $O(n^{100})$, ни за время $O(n^{1000})$, ни даже за время $O(n^{1\,000\,000})$.

На самом деле все еще хуже. Многие задачи — *тысячи* из них — имеют эту особенность: для них неизвестен алгоритм, который бы решал их для входных данных размером n за время $O(n^c)$ для какой-либо константы c, но и никто не доказал, что такой алгоритм не может существовать. Эти задачи возникают в самых разных областях, среди которых — логика, теория графов, арифметика и планирование.

Чтобы разочарование перешло на совершенно новый уровень, познакомьтесь с самым удивительным фактом: *если хотя бы для одной из этих задач существует алгоритм со временем работы $O(n^c)$, где c — некоторая константа, то алгоритм со временем работы $O(n^c)$ существует для каждой из них*. Мы называем эти задачи ***NP-полными***.

Алгоритм, который для входных данных размером n решает задачу за время $O(n^c)$, где c является константой, представляет собой ***алгоритм с полиномиальным временем работы***. Он называется так потому, что n^c с некоторым коэффициентом будет наиболее значащим членом в формуле для его времени работы. Для каждой NP-полной задачи известен алгоритм, решающий ее за время, не являющееся полиномиальным, но никто не доказал, что некоторые из NP-полных задач за полиномиальное время не разрешимы.

Есть разочарование еще большее: многие NP-полные задачи практически такие же, как и задачи, которые мы умеем решать за полиномиальное время. Отличие в условии просто мизерное. Например, вспомните из главы 6, "Кратчайшие пути", что алгоритм Беллмана–Форда находит кратчайший путь из одной вершины в ориентированном графе, даже если этот граф имеет ребра с отрицательным весом, за время $\Theta(nm)$ (n и m — соответственно количество вершин и ребер графа). Если граф задан списками смежности, то его входной размер $\Theta(n+m)$. Предположим, что $m \geq n$; тогда размер входных данных — $\Theta(m)$, и $nm \leq m^2$, так что время работы алгоритма Беллмана–Форда полиномиально зависит от размера входных данных (тот же результат получится и при $n > m$). Так что найти *кратчайшие* пути очень просто. Так что вас, наверное, удивит, что поиск длиннейшего ациклического пути (т.е. самого длинного пути без циклов) между двумя вершинами является NP-полной задачей. Более того, даже простое определение того, содержит ли граф путь без циклов не менее чем с заданным количеством ребер, является NP-полной задачей.

В качестве еще одного примера близких задач, где одна решается легко, а вторая является NP-полной, рассмотрим эйлеровы и гамильтоновы циклы. Обе эти задачи ищут пути в связных неориентированных графах (в ***неориентированном графе*** ребра не имеют направления, так что (u,v) и (v,u) представляют собой одно и то же ребро. Мы говорим, что ребро (u,v) ***инцидентно*** вершинам u и v. ***Связный граф*** имеет путь между каждой парой вершин. ***Эйлеров***[1] ***цикл*** начинается и заканчивается в одной и той же вершине и проходит по каждому ребру ровно один раз, хотя при этом может посещать вершины более одного раза. ***Гамильтонов***[2] ***цикл*** начинается и заканчивается в одной и той же вершине и посещает каждую вершину ровно один раз (за исключением, конечно, вершины, в которой цикл начинается и заканчивается). Если мы зададимся вопросом, имеет ли связанный неориентированный граф эйлеров цикл, алгоритм удивительно прост: надо определить ***степень*** каждой вершины (т.е. количество инцидентных ей ребер). Граф имеет эйлеров цикл тогда и только тогда, когда каждая его вершина имеет четную степень. Но если мы хотим узнать, содержит ли связанный неориентированный граф гамильтонов цикл, то это — NP-полная задача. Обратите внимание, что мы не ставим вопрос о порядке вершин гамильтонова цикла в графе и выясняем только, можно ли построить в этом графе гамильтонов цикл.

---

[1] Назван так в связи с доказательством в 1736 году Леонардом Эйлером (Leonard Euler) невозможности замкнутого обхода города Кенигсберга с проходом по всем семи его мостам ровно по одному разу.

[2] Назван так в честь Уильяма Гамильтона (William Hamilton), который в 1856 году описал математическую игру на графе, представляющем додекаэдр, в которой один игрок устанавливает пять фишек в пяти последовательных вершинах, а второй игрок должен завершить путь, образующий цикл, содержащий все вершины графа.

NP-полные задачи встречаются на удивление часто, и именно поэтому я включил в книгу посвященный им материал. Если вы пытаетесь найти алгоритм с полиномиальным временем для решения задачи, которая является NP-полной, вы, вероятно, не добьетесь ничего, кроме усталости и разочарования. Концепция NP-полных задач возникла примерно в начале 1970-х годов, но попытки решения задач, оказавшихся NP-полными (таких, как задача коммивояжера) были и задолго до этого. На сегодняшний день мы не знаем, существует ли алгоритм с полиномиальным временем работы для любой из NP-полных задач; но мы не можем и утверждать, что такой алгоритм не может существовать. Многие блестящие ученые потратили годы на поиски ответа на этот вопрос — без малейшего результата. Я не говорю, что вы не сможете найти полиномиальный алгоритм для решения NP-полной задачи, но я очень удивлюсь, если это у вас получится...

Классы P и NP и NP-полнота

В предыдущих главах меня волновало различие между временем работы $O(n^2)$ и $O(n \lg n)$. В этой же главе мы будем рады, если алгоритм имеет полиномиальное время работы, так что отличия $O(n^2)$ и $O(n \lg n)$ в этом случае рассматриваются как незначительные. Ученые в области алгоритмов обычно рассматривают задачи, решаемые алгоритмами с полиномиальным временем работы, как "легко решаемые". Если для решения задачи имеется алгоритм с полиномиальным временем работы, мы говорим, что эта задача принадлежит ***классу P***.

Сейчас вас может удивить, что мы считаем "легко решаемой" задачу, для решения которой требуется время $\Theta(n^{100})$. Для входных данных размером $n = 10$ не выглядит ли число 10^{100} слишком угрожающе? Да, конечно, ведь это целый гугол (googol, от которого произошло название "Google"). Но, к счастью, алгоритмы со временем работы $\Theta(n^{100})$ на практике не встречаются. Задачи из класса P, с которыми приходится иметь дело на практике, требуют для решения гораздо меньше времени. Я редко встречал алгоритмы с полиномиальным временем работы, худшим, чем $O(n^5)$. Кроме того, опыт показывает, что как только кто-то находит первый алгоритм с полиномиальным временем работы для некоторой задачи, тут же находятся другие, более эффективные алгоритмы. Так что, если бы кто-то разработал первый полиномиальный алгоритм со временем $\Theta(n^{100})$, имелись бы неплохие шансы, что нашлись бы и более быстрые решения.

Теперь предположим, что у нас есть предложенное решение задачи и вы хотите убедиться, что это решение является правильным. Например, в задаче о гамильтоновом цикле предлагаемое решение представляет собой последовательность вершин. Для того чтобы убедиться, что это решение является правильным, нужно проверить, что каждая вершина появляется в последовательности ровно один раз (за исключением совпадения первой и последний вершин), и если предложенная последовательность представляет собой $\langle v_1, v_2, v_3, \ldots, v_n, v_1 \rangle$, то граф должен содержать ребра (v_1, v_2), (v_2, v_3), (v_3, v_4), ..., (v_{n-1}, v_n) и (v_n, v_1). Можно легко убедиться в правильности предложенного решения задачи о гамильтоновом цикле за полиномиальное время. Если в общем случае можно проверить предложенное решение задачи за время, полиномиально зависящее от размера входных данных

задачи, то мы говорим, что эта задача принадлежит ***классу NP***[3]. Такое предлагаемое решение мы называем ***сертификатом***, и чтобы задача принадлежала классу NP, время проверки сертификата должно полиномиально зависеть от размера входных данных задачи и размера сертификата.

Если вы можете решить задачу за полиномиальное время, вы, конечно же, сможете проверить сертификат этой задачи за полиномиальное время. Другими словами, каждая задача из класса P автоматически принадлежит классу NP. Обратное утверждение — что все задачи из класса NP принадлежат также классу P — представляет собой вопрос, который многие годы ставит в тупик ученых. Его часто называют "проблема P = NP?".

NP-полные задачи являются "наитруднейшими" в классе NP. Неформально говоря, задача является ***NP-полной***, если она удовлетворяет двум условиям: (1) принадлежит классу NP и (2) в случае, если для этой задачи существует алгоритм с полиномиальным временем работы, имеется способ преобразования *каждой* задачи из класса NP в эту задачу таким образом, чтобы все они решались за полиномиальное время. Если алгоритм с полиномиальным временем работы имеется для *любой* NP-полной задачи — т.е. если любая NP-полная задача принадлежит классу P, — то тогда P = NP. Поскольку NP-полные задачи являются самыми сложными в классе NP, если окажется, что какая-то задача из класса NP является неразрешимой за полиномиальное время, то не разрешима за полиномиальное время ни одна из NP-полных задач. Задача называется ***NP-сложной***, если она удовлетворяет второму условию NP-полноты, но может как входить в класс NP, так и не входить в него.

Вот для удобства список соответствующих определений.

- **P**: задачи разрешимы за полиномиальное время, т.е. мы можем решить задачу за время, полиномиально зависящее от размера входных данных.
- **Сертификат**: предлагаемое решение задачи.
- **NP**: задачи, проверяемые за полиномиальное время, т.е. если для такой задачи имеется сертификат, мы можем убедиться, что он представляет собой решение задачи, за время, полиномиально зависящее от размера входных данных.
- **NP-сложная задача**: задача, такая, что если существует алгоритм ее решения за полиномиальное время, то любая задача в NP может быть преобразована в данную задачу таким образом, что все задачи из NP решаются за полиномиальное время.
- **NP-полная задача**: задача, являющаяся NP-сложной и принадлежащая классу NP.

Задачи принятия решения и приведения

Говоря о классах P и NP или о концепции NP-полноты, мы ограничиваемся ***задачами принятия решения***: их выход представляет собой один бит, указывающий ответ "да" или

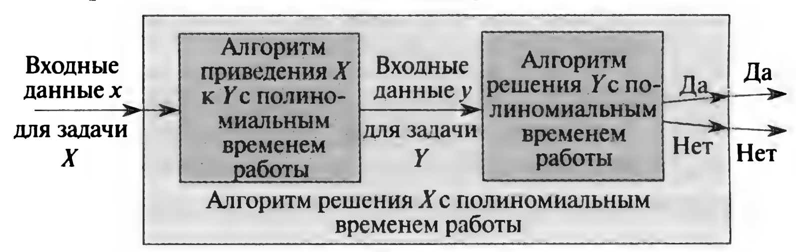
---

[3] *Вы, вероятно, уже предположили, что название P происходит от "полиномиальное время". Если вам интересно, откуда взялось название NP, то оно происходит от термина "недетерминированное полиномиальное время". Это эквивалентный, хотя и менее интуитивный, способ рассмотрения этого класса задач.*

"нет". Я сформулировал задачи поиска эйлерова и гамильтонова циклов как "содержит ли граф эйлеров цикл?" и "содержит ли граф гамильтонов цикл?"

Однако некоторые задачи являются задачами оптимизации, в которых мы хотим найти наилучшие возможные решения, а не получить ответ "да" или "нет". К счастью, зачастую достаточно легко перебросить мост через эту пропасть путем переформулирования задачи оптимизации как задачи принятия решения. Например, рассмотрим задачу поиска кратчайшего пути. Для ее решения мы использовали алгоритм Беллмана–Форда. Каким образом представить задачу поиска кратчайшего пути как задачу принятия решения? Можно спросить "содержит ли граф путь между двумя указанными вершинами, вес которого не превышает заданное значение k?" Мы не просим указать вершины или ребра этого пути, мы просто выясняем его наличие. Предполагая, что вес пути является целым числом, мы можем найти фактический вес кратчайшего пути между двумя вершинами, задавая вопросы с ответами "да/нет". Каким образом? Зададим вопрос о существовании пути для $k = 1$. Если ответ на него — "нет", то зададим вопрос с $k = 2$. Если ответ — "нет", попробуем $k = 4$. Будем удваивать значение k до тех пор, пока не получим ответ "да". Если это последнее проверенное значение k равно k', значит, ответ находится где-то между $k'/2$ и k'. Точное значение можно найти методом бинарного поиска в интервале от $k'/2$ до k'. Такой подход не скажет нам, какие вершины и ребра находятся на кратчайшем пути, но как минимум сообщит, есть ли такой путь.

Второе условие NP-полноты задачи требует, чтобы при наличии алгоритма ее решения с полиномиальным временем работы существовал способ преобразовать каждую задачу из NP в эту задачу таким образом, чтобы их все можно было решить за полиномиальное время. Ограничившись задачами принятия решения, давайте рассмотрим общую идею преобразования одной задачи принятия решения X в другую задачу принятия решения Y так, чтобы если существует алгоритм с полиномиальным временем работы для решения Y, то есть алгоритм с полиномиальным временем работы и для задачи X. Мы называем такое преобразование ***приведением***. Вот основная идея приведения.



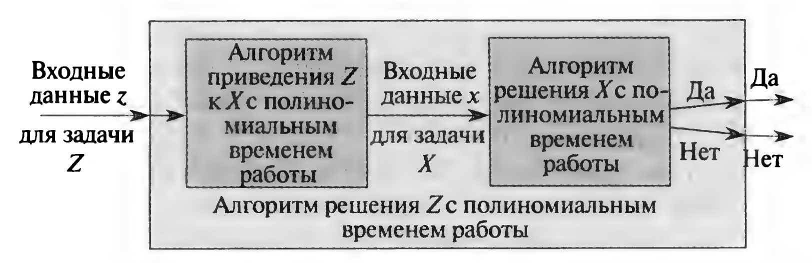
У нас имеются некоторые входные данные x размером n для задачи X. Мы преобразуем эти входные данные во входные данные y для задачи Y и делаем это за время, полиномиально зависящее от n, скажем, за $O(n^c)$ для некоторой константы c. Способ преобразования входных данных x во входные данные y обладает важным свойством: если алгоритм Y принимает решение "да" для входных данных y, то алгоритм X должен принимать решение "да" для входных данных x; если же Y решает "нет" для входных данных y, то X выдает "нет" для входных данных x. Это преобразование мы называем ***алгоритмом приведения с полиномиальным временем работы***. Давайте посмотрим, сколько времени требуется

алгоритму для решения задачи X. Алгоритм приведения требует времени $O(n^c)$, причем его вывод не может быть больше, чем время работы, так что размер выходных данных алгоритма приведения равен $O(n^c)$. Но эти выходные данные являются входными данными y для алгоритма решения задачи Y. Поскольку алгоритм решения задачи Y является алгоритмом с полиномиальным временем работы, для входных данных размером m он выполняется за время $O(m^d)$ для некоторой константы d. Но здесь m представляет собой $O(n^c)$, а потому алгоритм решения задачи Y выполняется за время $O((n^c)^d) = O(n^{cd})$. Поскольку и c, и d являются константами, константой является и cd, так что алгоритм для решения задачи Y является алгоритмом с полиномиальным временем работы. Общее время решения задачи X составляет $O(n^c + n^{cd})$, так что она также решается за полиномиальное время.

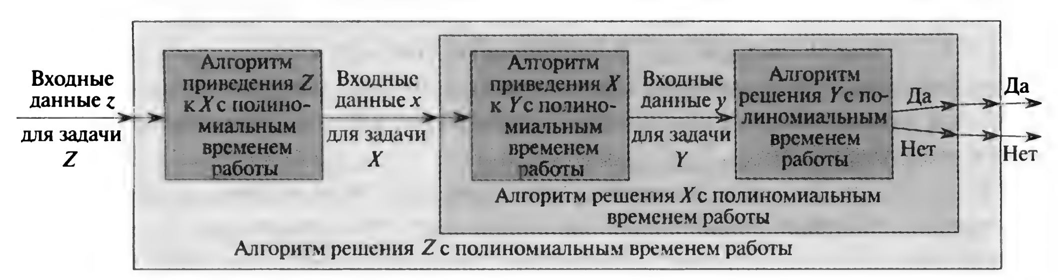
Этот подход показывает, что если задача Y — "легкая" (разрешима за полиномиальное время), то таковой же является и задача X. Но мы будем использовать приведение за полиномиальное время, чтобы демонстрировать не легкость, а сложность задач.

> Если задача X является NP-сложной и мы можем привести ее к задаче Y за полиномиальное время, то задача Y также является NP-сложной.

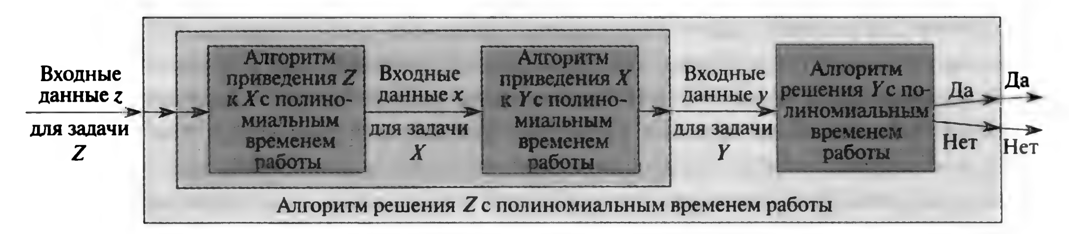
Почему это утверждение должно выполняться? Давайте предположим, что задача X является NP-сложной и что есть алгоритм приведения с полиномиальным временем работы, преобразующий входные данные X во входные данные Y. Поскольку X является NP-сложной задачей, есть способ для преобразования любой задачи, скажем, Z, из класса NP в X, такой, что если X имеет алгоритм для ее решения за полиномиальное время, то это справедливо и в отношении задачи Z. Теперь вы знаете, как выполняется такое преобразование.



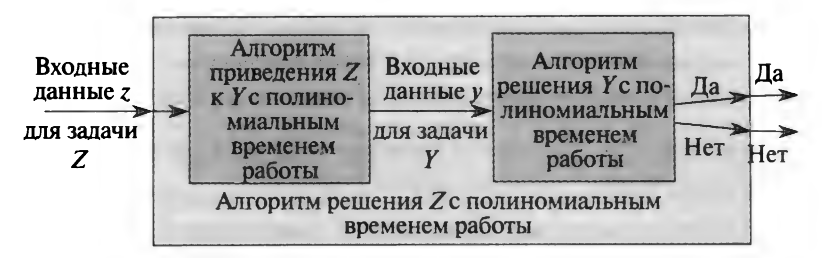
Поскольку мы можем преобразовать входные данные задачи X во входные данные задачи Y за полиномиальное время, можно воспользоваться ранее выполненным приведением X.



Вместо группирования в единое целое приведения задачи X к задаче Y за полиномиальное время и решение задачи Y, сгруппируем два приведения за полиномиальное время в одно.



Заметим, что если немедленно после приведения за полиномиальное время задачи Z к задаче X выполнить приведение за полиномиальное время задачи X к задаче Y, мы получим приведение задачи Z к задаче Y за полиномиальное время.



Просто для того, чтобы убедиться, что два приведения за полиномиальное время вместе составляют одно такое приведение, выполним анализ, аналогичный приведенному ранее. Предположим, что входные данные z для задачи Z имеют размер n, что приведение Z к X занимает время $O\left(n^c\right)$ и что приведение X к Y для входных данных размером m выполняется за время $O\left(m^d\right)$, где c и d — константы. Выходные данные приведения Z к X не могут быть больше, чем затраченное время, так что эти выходные данные, которые одновременно являются входными данными x приведения X к Y, имеют размер $O\left(n^c\right)$. Теперь мы знаем, что размер m входных данных приведения X к Y равен $m = O\left(n^c\right)$, а потому время приведения X к Y представляет собой $O\left(\left(n^c\right)^d\right) = O\left(n^{cd}\right)$. Поскольку и c, и d являются константами, это второе приведение выполняется за полиномиальное время от n.

Кроме того время, затраченное на последнем этапе (полиномиальном алгоритме решения задачи Y), также полиномиально зависит от n. Предположим что алгоритм решения задачи Y для входных данных размером p выполняется за время $O\left(p^b\right)$, где b — константа. Как и ранее, выходные данные приведения не могут превышать время их генерации, так что $p = O\left(n^{cd}\right)$, а это означает, что алгоритм решения задачи Y выполняется за время $O\left(\left(n^{cd}\right)^b\right) = O\left(n^{bcd}\right)$. Поскольку b, c и d являются константами, алгоритм решения задачи Y выполняется за время, полиномиально зависящее от размера исходных данных n. Так что алгоритм решения задачи Z выполняется за время $O\left(n^c + n^{cd} + n^{bcd}\right)$, являющееся полиномиально зависящим от n.

Итак, что же мы только что видели? Мы показали, что если задача X является NP-сложной и существует приведение с полиномиальным временем работы, которое преобразует входные данные x задачи X во входные данные y задачи Y, то задача Y также является NP-сложной. То, что задача X NP-сложная, означает, что все задачи из класса NP приводятся к ней за полиномиальное время, позволяет выбрать произвольную задачу Z из

NP, которая приводится к X за полиномиальное время, и тем самым показать, что она за полиномиальное время приводится и к Y.

Наша конечная цель заключается в демонстрации NP-полноты задач. Теперь для того, чтобы показать, что задача Y NP-полная, нам достаточно

- показать, что она принадлежит классу NP (для чего достаточно показать, что имеется способ проверки сертификата для Y за полиномиальное время),
- взяв некоторую задачу X, NP-сложность которой нам известна, показать, что она приводится к задаче Y за полиномиальное время.

Есть еще одна небольшая деталь, о которой я пока ничего не сказал: первичная задача. Нам нужно начать с некоторой NP-полной задачи M, к которой за полиномиальное время приводится каждая задача из класса NP. После этого мы сможем приводить M к другим задачам за полиномиальное время, чтобы показать, что эти другие задачи являются NP-сложными, эти другие задачи приводить за полиномиальное время к очередным задачам для того, чтобы показать, что те также являются NP-сложными, и т.д. Имейте также в виду, что нет никаких ограничений на количество других задач, которые могут быть приведены к одной, так что генеалогическое дерево NP-полных задач начинается с первичной задачи, а затем разветвляется.

Первичная задача

В разных книгах первичные задачи различны. Это нормально, поскольку после того, как вы приведете одну первичную задачу к другой, эта другая задача также может выступать в роли первичной. Одной из часто встречающихся первичных задач является задача выполнимости булевой формулы. Я кратко опишу эту задачу, но не буду доказывать, что за полиномиальное время к ней сводится любая задача из класса NP. Это доказательство весьма длинное и, осмелюсь сказать, весьма утомительное.

Начну с того, что "булева" — это математический сленг для простой логики, в которой переменные могут принимать только два значения — 0 и 1 (именуемые булевыми значениями), а операторы принимают одно или два булевых значения и выдают также булево значение. Мы уже встречались с операцией исключающего или (XOR) в главе 8, "Основы криптографии". Типичными булевыми операторами являются операции "и" (AND), "или" (OR), "не" (NOT), "следует" (IMPLIES) и "эквивалентно" (IFF).

- x AND y равно 1, только если и x, и y равны 1; в противном случае (когда хотя бы одно из значений равно 0) x AND y равно 0.
- x OR y равно 0, только если и x, и y равны 0; в противном случае (когда хотя бы одно из значений равно 1) x OR y равно 1.
- NOT x представляет собой противоположное x значение: 0, если $x = 1$, и 1, если $x = 0$.
- x IMPLIES y равно 0, только если $x = 1$ и $y = 0$; в противном случае (либо $x = 0$, либо и x, и y равны 1) x IMPLIES y равно 1.

- x IFF y означает "x тогда и только тогда, когда y", и это значение равно 1, только если и x, и y одновременно равны 0 или одновременно равны 1. Если $x \neq y$, то x IFF $y = 0$.

Имеется 16 возможных булевых бинарных (получающих два операнда) операторов; здесь показаны только наиболее распространенные из них[4]. ***Булева формула*** состоит из булевых переменных, операторов и скобок для их группировки.

В ***задаче о выполнимости булевой формулы*** входные данные представляют собой булеву формулу, а поставленный вопрос — существует ли такой набор значений переменных, чтобы выполненные по формуле вычисления давали значение 1. Если такой набор существует, мы говорим, что формула выполнима. Например, булева формула

$$\big((w \text{ IMPLIES } x) \text{ OR NOT } \big(((\text{NOT } w) \text{ IFF } y) \text{ OR } z\big)\big) \text{ AND } (\text{NOT } x)$$

выполнима: пусть $w = 0$, $x = 0$, $y = 1$ и $z = 1$. Тогда формула вычисляется следующим образом:

$$\begin{aligned} &\big((0 \text{ IMPLIES } 0) \text{ OR NOT } \big(((\text{NOT } 0) \text{ IFF } 1) \text{ OR } 1\big)\big) \text{ AND } (\text{NOT } 0) \\ &= \big(1 \text{ OR NOT } ((1 \text{ IFF } 1) \text{ OR } 1)\big) \text{ AND } 1 \\ &= \big(1 \text{ OR NOT } (1 \text{ OR } 1)\big) \text{ AND } 1 \\ &= (1 \text{ OR } 0) \text{ AND } 1 \\ &= 1 \text{ AND } 1 \\ &= 1. \end{aligned}$$

С другой стороны, следующая простая формула невыполнима: x AND (NOT x).
Если $x = 0$, эта формула вычисляется как 0 AND 1, что равно 0; если же $x = 1$, мы получаем 1 AND 0, что, опять же, равно 0.

Сборник NP-полных задач

Рассмотрим, с задачей выполнимости булевой формулы в качестве первичной, некоторые из задач, NP-полноту которых можно показать путем приведения за полиномиальное время. Вот генеалогическое дерево приведений, которые я имею в виду.

Я не буду показывать все приведения в этом дереве, потому что некоторые из них довольно громоздки и утомительны. Но мы рассмотрим пару из них, интересных тем, что они показывают приведение задачи из одной предметной области к задаче из совершенно другой, например логику (3-CNF-выполнимость) к графам (задача о клике).

3-CNF-выполнимость

Поскольку булевы формулы могут содержать любой из 16 бинарных булевых операторов, а также поскольку они могут быть размещены в скобках произвольным образом,

---

[4] *Некоторые из этих 16 операторов совершенно неинтересны, как, например, оператор, возвращающий значение 0 независимо от значений операндов.*



весьма трудно выполнить приведение непосредственно задачи выполнимости булевой формулы, играющей роль первичной задачи. Вместо этого мы определим родственную задачу, которая также состоит в определении выполнимости булевых формул, но при этом имеет некоторые ограничения на структуру формулы, являющейся входными данными. Выполнять приведение этой задачи будет гораздо проще. Итак, потребуем, чтобы формула представляла собой набор операторов AND, примененных к выражениям в скобках, где каждое такое выражение представляет собой применение операторов OR к трем членам, причем каждый член является литералом, т.е. либо переменной, либо ее отрицанием. Булева формула в этом виде называется представленной в ***3-коньюнктивной нормальной форме***, или ***3-CNF***. Например, булева формула

$$\left(w \text{ OR } (\text{NOT } w) \text{ OR } (\text{NOT } x)\right) \text{ AND } \left(y \text{ OR } x \text{ OR } z\right)$$
$$\text{AND } \left((\text{ AND } w) \text{ OR } (\text{NOT } y) \text{ OR } (\text{NOT } z)\right)$$

является 3-CNF-формулой. Ее первым выражением является $\left(w \text{ OR } (\text{NOT } w) \text{ OR } (\text{NOT } x)\right)$.

Задача выяснения, имеет ли булева 3-CNF-формула набор выполняющих ее переменных, — ***задача 3-CNF-выполнимости***, — является NP-полной. Ее сертификат представляет собой предлагаемое назначение значений 0 и 1 переменным. Проверка сертификата проста: надо просто присвоить назначенные значения переменным и, вычислив формулу, убедиться, что в результате вычисления получилось значение 1. Чтобы показать, что задача 3-CNF-выполнимости является NP-сложной, мы приведем к ней задачу о выполнимости булевой формулы (без каких-либо ограничений). Я вновь не буду вдаваться в (не столь уж интересные) подробности. Гораздо интереснее будет посмотреть на приведение задачи из одной предметной области к задаче из другой области, чем мы и собираемся заняться.

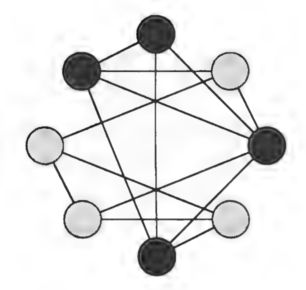
А вот и первое расстройство: хотя упомянутая задача 3-CNF-выполнимости NP-полная, имеется алгоритм с полиномиальным временем работы, определяющий, является ли выполнимой 2-CNF-формула (такая же, как и 3-CNF-формула, с тем отличием, что она имеет по два, а не по три литерала в каждом выражении в скобках). Такое небольшое изменение — и такое усложнение решения!

Клика

Теперь мы рассмотрим интересное приведение для задач из различных предметных областей: задачи 3-CNF-выполнимости к неориентированным графам. ***Клика*** в неориентированном графе G — это подмножество S вершин графа, такое, что в графе имеются ребра, соединяющие каждую пару вершин в S. ***Размер клики*** равен количеству вершин, которые она содержит.

Как вы догадываетесь, клики играют роль в теории социальных сетей. Если моделировать каждого человека как вершину, а отношения между людьми — как неориентированные ребра, то клики представляют собой группу лиц, в которой все имеют отношения друг с другом. Клики также применяются в области биоинформатики, техники и химии.

Задача о клике получает на вход граф G и положительное целое число k, и в задаче спрашивается, содержит ли граф G клику размером k. Например, приведенный ниже граф имеет клику размером 4, вершины которой выделены темной штриховкой. В этом графе нет других клик размером 4 или больше.



Проверить сертификат очень просто. Сертификат представляет собой список k вершин, о которых утверждается, что они образуют клику, так что мы просто должны проверить, что каждая из k вершин имеет ребра к другим $k-1$ вершинам. Эта проверка легко выполнима за время, полиномиально зависящее от размера графа. Так что мы знаем, что задача о клике принадлежит классу NP.

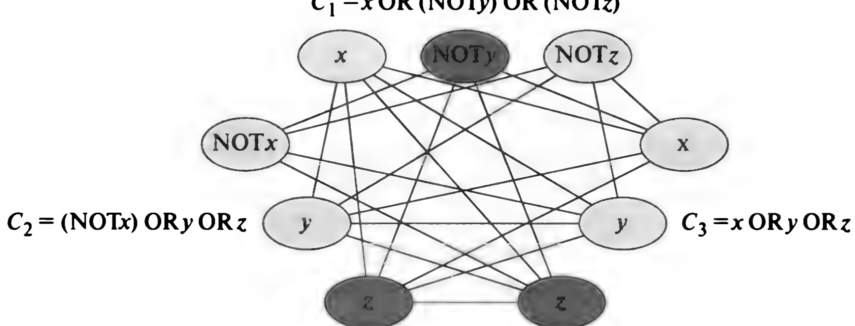
Но как же привести задачу о выполнимости булевой формулы к задаче, связанной с графом? Начнем с булевой 3-CNF-формулы. Предположим, что формула имеет вид C_1 AND C_2 AND C_3 AND $\cdots$ AND C_k, где каждое C_r представляет собой одно из k выражений. На основе этой формулы мы за полиномиальное время построим граф, который будет иметь k клик тогда и только тогда, когда 3-CNF-формула будет выполнимой. Мы должны обеспечить три вещи: построение графа, доказательство, что это построение выполняется за время, полиномиально зависящее от размера 3-CNF-формулы, и доказательство того, что граф имеет k клик в том и только в том случае, если существует некоторый способ назначения значений переменным в 3-CNF-формуле таким образом, что ее значение оказывается равным 1.

Для построения графа из 3-CNF-формулы сосредоточимся на r-м выражении C_r. Оно состоит из трех литералов; назовем их l_1^r, l_2^r и l_3^r, так что $C_r = l_1^r$ OR l_2^r OR l_3^r. Каждый литерал является либо переменной, либо ее отрицанием. Для каждого литерала мы создаем одну вершину, так что для выражения C_r мы создадим тройку вершин: v_1^r, v_2^r и v_3^r. Мы добавляем ребро между вершинами v_i^r и v_j^s, если выполняются два условия:

- v_i^r и v_j^s находятся в разных тройках, т.е. r и s являются номерами разных выражений,
- их литералы не являются отрицанием один другого.

Например, показанный ниже граф соответствует 3-CNF-формуле

$$(x \text{ OR } (\text{NOT } y) \text{ OR } (\text{NOT } z)) \text{ AND } ((\text{NOT } x) \text{ OR } y \text{ OR } z) \text{ AND } (x \text{ OR } y \text{ OR } z).$$



Достаточно легко показать, что это приведение может быть выполнено за полиномиальное время. Если 3-CNF-формула содержит k выражений, то в ней $3k$ литералов, так что всего граф имеет $3k$ вершин. Каждая вершина имеет не более $3k-1$ ребер ко всем остальным $3k-1$ вершинам, так что общее количество ребер не превышает $3k(3k-1) = 9k^2 - 3k$. Размер построенного графа полиномиально зависит от размера входной 3-CNF-формулы, и очень легко определить, какие ребра имеются в графе.

Наконец нужно показать, что построенный граф имеет клику размером k тогда и только тогда, когда 3-CNF-формула выполнима. Начнем с предположения, что формула выполнима, и покажем, что граф имеет k-клику. Если существует удовлетворяющий набор значений переменных, каждое выражение C_r содержит по крайней мере один литерала l_i^r, имеющий значение 1, и каждый такой литерал соответствует вершине v_i^r в графе. Если мы выберем по одному такому литералу из каждого из k выражений, то получим соответствующее множество S из k вершин. Я утверждаю, что S является k-кликой. Рассмотрим любые две вершины в S. Они соответствуют литералам в разных выражениях, которые дают 1 при использовании выполняющего набора значений. Эти литералы не могут быть отрицаниями один другого, поскольку в таком случае один из них был бы единицей, а другой — нулем. Но поскольку эти литералы не являются отрицанием один другого, между соответствующими вершинами при построении графа было добавлено ребро. Поскольку в качестве рассматриваемой пары мы можем выбрать любые две вершины в S, мы видим, что между всеми парами вершин в S имеются ребра. Следовательно, множество S из k вершин представляет собой k-клику.

Теперь следует рассмотреть ситуацию в обратном направлении: если граф имеет k-клику S, то 3-CNF-формула выполнима. В графе нет ребер между вершинами из одной тройки, а значит, S содержит ровно одну вершину из каждой тройки. Для каждой вершины v_i^r из S назначим соответствующему литералу l_i^r в 3-CNF-формуле единичное значение. Мы не должны беспокоиться о присвоении единицы и литералу, и его отрицанию, так как

k-клика не может одновременно содержать вершины, соответствующие литералу и его отрицанию. Поскольку каждое выражение имеет литерал, равный 1, каждое выражение выполнимо, а потому выполнима и вся 3-CNF-формула. Значения переменным, не соответствующим вершинам клики, присваиваются произвольным образом; они не влияют на выполнимость формулы.

В приведенном выше примере выполняющим набором являются $y = 0$ и $z = 1$; значение x не играет роли. Получающаяся 3-клика состоит из заштрихованных вершин, которые соответствуют NOT y из C_1 и z из C_2 и C_3.

Таким образом, мы показали, что существует приведение за полиномиальное время NP-полной задачи 3-CNF-выполнимости проблемы к поиску k-клики. Если имеется булева 3-CNF-формула с k выражениями и вам надо найти для нее выполняющий набор значений переменных, то можно использовать описанное выше построение для преобразования за полиномиальное время формулы в неориентированный граф и определить, имеет ли этот граф k-клику. Если бы можно было за полиномиальное время определить наличие k-клики в графе, то тем самым за полиномиальное время можно было бы определить, имеет ли 3-CNF-формула выполняющий набор. Поскольку задача о 3-CNF-выполнимости является NP-полной, таковой же является и задача определения, содержит ли граф k-клику. В качестве бесплатного приложения, если бы вы могли определить не только наличие в графе k-клики, но и составляющие ее вершины, вы могли бы использовать эту информацию для поиска выполняющего набора значений для 3-CNF-формулы.

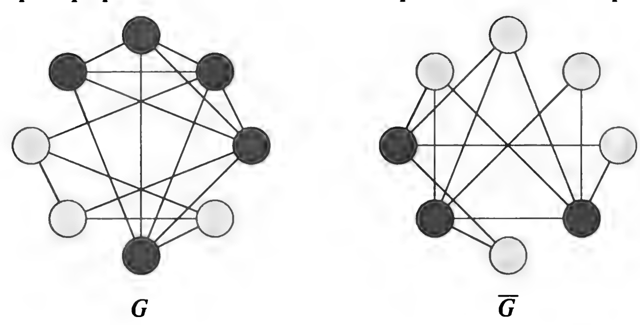
Вершинное покрытие

Вершинное покрытие неориентированного графа G представляет собой подмножество S его вершин, такое, что каждое ребро G инцидентно по крайней мере одной вершине из S. Мы говорим, что каждая вершина в S "покрывает" инцидентные ей ребра. ***Размер вершинного покрытия*** равен количеству содержащихся в нем вершин. Как и задача о клике, ***задача о вершинном покрытии*** получает в качестве входных данных неориентированный граф G и положительное целое число m. В задаче спрашивается, имеет ли граф G вершинное покрытие размером m. Как и задача о клике, задача о вершинном покрытии имеет приложения в биоинформатике. Еще в одной задаче у вас есть здание с коридорами и камеры на перекрестках этих коридоров, которые могут полностью сканировать последние; требуется определить, достаточно ли m камер для полного сканирования всех коридоров. Здесь ребра моделируют коридоры, а вершины — их пересечения. В еще одном приложении эта задача помогает в разработке стратегий по предотвращению атак компьютерных сетей червем.

Очевидно, что сертификатом для задачи вершинного покрытия является список предлагаемых вершин. За полиномиальное время легко убедиться, что предлагаемое вершинное покрытие имеет размер m и действительно охватывают все ребра; таким образом, мы видим, что эта задача принадлежит классу NP.

Генеалогическое дерево NP-полноты на с. 185 говорит нам, что мы будем приводить задачу о клике к задаче о вершинном покрытии. Предположим, что входными данными для задачи о клике являются неориентированный граф G с n вершинами и положительное целое число k. За полиномиальное время мы сгенерируем входной граф $\overline{G}$ для задачи о

вершинном покрытии, такой, что граф G имеет клику размером k тогда и только тогда, когда граф $\overline{G}$ имеет вершинное покрытие размером $n-k$. Это очень простое приведение. Граф $\overline{G}$ имеет те же вершины, что и граф G, и ребра, отсутствующие в G. Иными словами, ребро (u,v) имеется в графе $\overline{G}$ тогда и только тогда, когда ребра (u,v) в графе G *нет*. Возможно, вы решите, что вершинное покрытие размером $n-k$ в $\overline{G}$ состоит из вершин, которые *не* входят в клику из k вершин в графе G, — и вы будете совершенно правы! Ниже приведены примеры графов G и $\overline{G}$ с восемью вершинами. Пять вершин, образующих



клику в G, и остальные три вершины, образующие вершинное покрытие в $\overline{G}$, выделены на рисунке темной штриховкой. Обратите внимание, что каждое ребро в $\overline{G}$ инцидентно как минимум одной заштрихованной вершине.

Нам нужно показать, что G имеет k-клику тогда и только тогда, когда $\overline{G}$ имеет вершинное покрытие размером $n-k$. Для начала предположим, что G имеет k-клику C. Пусть S состоит из $n-k$ вершин, не входящих в C. Я утверждаю, что каждое ребро в $\overline{G}$ инцидентно по крайней мере одной вершине из S. Пусть (u,v) — произвольное ребро в $\overline{G}$. Оно принадлежит $\overline{G}$, поскольку не принадлежит G. А так как (u,v) не принадлежит G, как минимум одна из вершин u и v находится не в клике C графа G, поскольку ребро соединяет все пары вершин в C. Так как по крайней мере одна из вершин u и v находится не в клике C, как минимум одна из них находится в S, что означает, что ребро (u,v) инцидентно по меньшей мере одной из вершин в S. Поскольку мы выбрали (u,v) как произвольное ребро из $\overline{G}$, мы видим, что S является вершинным покрытием графа $\overline{G}$.

Теперь пойдем в обратном направлении. Предположим, что $\overline{G}$ имеет вершинное покрытие S, содержащее $n-k$ вершин, и пусть C состоит из k вершин, не входящих в S. Каждое ребро в $\overline{G}$ инцидентно некоторой вершине в S. Другими словами, если (u,v) является ребром в $\overline{G}$, то по крайней мере одна из вершин u и v находится в S. Если вы вспомните определение контрапозиции на с. 33, то увидите, что контрапозицией данного следствия является утверждение, что если ни u, ни v не находятся в S, то (u,v) не входит в $\overline{G}$, а следовательно, (u,v) принадлежит G. Другими словами, если и u, и v входят в C, то ребро (u,v) присутствует в G. Так как u и v — произвольная пара вершин из C, мы видим, что в G имеются ребра между всеми парами вершин в C, т.е. C является k-кликой.

Таким образом, мы показали, что существует приведение за полиномиальное время NP-полной задачи определения, содержит ли неориентированный граф k-клику, к задаче выяснения, содержит ли неориентированный граф вершинное покрытие размером $n-k$. Если у вас имеется неориентированный граф G и вы хотите знать, содержит ли он k-клику,

можно воспользоваться только что описанным построением и преобразовать за полиномиальное время граф G в граф $\overline{G}$ и выяснить, содержит ли граф $\overline{G}$ вершинное покрытие с $n-k$ вершинами. Если за полиномиальное время можно определить, имеет ли $\overline{G}$ вершинное покрытие размером $n-k$, то таким образом можно определить за полиномиальное время, имеет ли граф G k-клику. Поскольку задача о клике является NP-полной, таковой же является и задача о вершинном покрытии. В качестве бесплатного приложения, если бы вы могли определить не только наличие у графа $\overline{G}$ вершинного покрытия из $n-k$ вершин, но и сами эти вершины, эту информацию можно было бы использовать для поиска вершин, составляющих k-клику.

Гамильтонов цикл и гамильтонов путь

Мы уже встречались с задачей о гамильтоновом цикле: содержит ли связный неориентированный граф гамильтонов цикла (путь, который начинается и заканчивается в одной и той же вершине и посещает все прочие вершины ровно один раз)? Непосредственные приложения этой задачи не совсем ясны, но из генеалогического дерева NP-полноты на с. 185 можно увидеть, что эта задача используется для доказательства того, что задача коммивояжера является NP-полной (а практичность этой задачи, как мы уже видели, сомнений не вызывает).

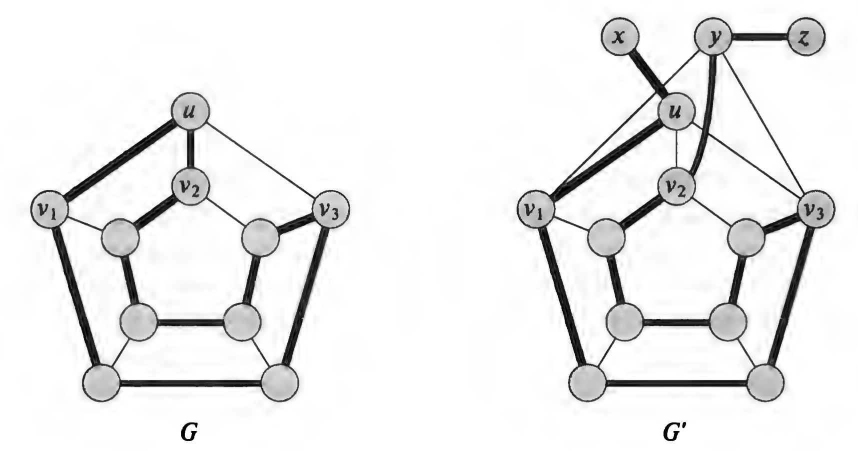
С этой задачей тесно связана задача о гамильтоновом пути, в которой спрашивается, содержит ли граф путь, который посещает все вершины ровно один раз, но не требуется, чтобы этот путь был циклом. Эта задача также является NP-полной, и мы воспользуемся ею, чтобы показать, что NP-полной является и задача о наидлиннейшем ациклическом пути.

Сертификат для обеих гамильтоновых задач тривиален: это упорядоченный список вершин в гамильтоновом цикле или пути (для гамильтонова цикла повторять первую вершину в конце не требуется). Для проверки сертификата нужно только проверить, что каждая вершина появляется в списке ровно один раз и что граф содержит ребра между каждой парой смежных вершин в списке. В случае задачи о гамильтоновом цикле надо также убедиться в наличии ребра между первой и последней вершинами в списке.

Я не буду подробно останавливаться на приведении за полиномиальное время задачи о вершинном покрытии к задаче о гамильтоновом цикле, которое показывает, что последняя является NP-сложной. Оно довольно сложное и опирается на так называемый виджет (widget) — часть графа, которая обеспечивает определенные свойства. Виджет, используемый в данном приведении, обладает тем свойством, что любой гамильтонов цикл в графе, построенном приведением, может пройти через виджет только одним из трех способов.

Чтобы привести задачу о гамильтоновом цикле к задаче о гамильтоновом пути, мы начинаем со связного неориентированного графа G с n вершинами и образуем из него новый связный неориентированный граф G' с $n+3$ вершинами. Мы выбираем любую вершину u в графе G. Пусть смежные с ней вершины — $v_1, v_2, \ldots, v_k$. Чтобы построить граф G', мы добавляем три новые вершины x, y и z, а также ребра (u,x) и (y,z) наряду с ребрами (v_1,y), (v_2,y), ..., (v_k,y) между y и всеми вершинами, смежными с u. Вот пример такого построения.

(v_1, y), (v_2, y), ..., (v_k, y) между y и всеми вершинами, смежными с u. Вот пример такого построения.



Заштрихованные ребра указывают гамильтонов цикл в графе G и соответствующий гамильтонов путь в графе G'. Это приведение выполняется за полиномиальное время, так как G' содержит только на три вершины больше, чем граф G, и не более чем $n+1$ дополнительное ребро.

Как обычно, следует показать, что приведение работает, т.е. что граф G имеет гамильтонов цикл тогда и только тогда, когда граф G' имеет гамильтонов путь. Предположим, что граф G имеет гамильтонов цикл. Он должен содержать ребро (u, v_i) для некоторой вершины v_i, смежной с вершиной u, а следовательно, смежной с y в графе G'. Чтобы сформировать гамильтонов путь в графе G, перейдем от x к z, беря все ребра гамильтонова цикла, кроме (u, v_i), и добавляя ребра (u, x), (v_i, y) и (y, z). В приведенном выше примере v_i представляет собой вершину v_2, так что из гамильтонова пути удаляется ребро (v_2, u) и добавляются ребра (u, x), (v_2, y) и (y, z).

Теперь предположим, что граф G' имеет гамильтонов путь. Поскольку вершины x и z имеют только по одному инцидентному ребру, гамильтонов путь должен идти от x к z и должен содержать ребро (v_i, y) для некоторой смежной с y вершины, а следовательно, смежной с u. Чтобы найти гамильтонов цикл в графе G, удалим x, y и z и все инцидентные им ребра и воспользуемся всеми ребрами гамильтонова пути в графе G' вместе с ребром (v_i, u).

Завершение такое же, как и для всех рассмотренных ранее приведений. Существует приведение за полиномиальное время NP-полной задачи определения, содержит ли связный неориентированный граф гамильтонов цикл, к задаче определения, содержит ли связный неориентированный граф гамильтонов путь. Так как первая задача NP-полная, таковой же является и последняя. Кроме того, знание ребер гамильтонова пути дает ребра гамильтонова цикла.

Задача коммивояжера

В версии принятия решения ***задачи коммивояжера*** задан полный неориентированный граф с неотрицательными целыми весами ребер и неотрицательное целое число k. ***Полный граф*** имеет ребра между каждой парой вершин, так что если в нем имеется n вершин, то он содержит $n(n-1)$ ребер. Требуется выяснить, имеет ли граф цикл, который содержит все вершины и общий вес которого не превышает k.

Очень легко показать, что эта задача принадлежит классу NP. Сертификат представляет собой упорядоченный список вершин цикла. Можно легко проверить за полиномиальное время, посещает ли цикл все вершины и имеет ли он общий вес, не превышающий k.

Чтобы показать, что задача коммивояжера является NP-сложной, мы выполним очень простое приведение к ней задачи о гамильтоновом цикле. Для данного графа G, являющегося входным для задачи о гамильтоновом цикле, мы строим полный граф G' с теми же вершинами, что и у графа G. Вес ребра (u,v) в графе G' устанавливается равным 0, если (u,v) является ребром графа G, и равным 1, если (u,v) не входит в G. Устанавливаем $k=0$. Такое приведение выполняется за время, полиномиально зависящее от размера графа G, поскольку оно добавляет в граф не более $n(n-1)$ ребер.

Чтобы показать, что это приведение работает, нам нужно показать, что граф G имеет гамильтонов цикл тогда и только тогда, когда граф G' имеет цикл с нулевым весом, включающий все вершины. Это легко сделать. Предположим, что граф G имеет гамильтонов цикл. Тогда каждое ребро цикла находится в G, так что каждое из этих ребер в графе G' имеет нулевой вес. Таким образом, граф G' имеет цикл, содержащий все вершины, и общий вес этого цикла нулевой. И обратно, предположим, что граф G' имеет цикл, содержащий все вершины, и общий вес этого цикла нулевой. Тогда каждое ребро этого цикла должно иметься и в графе G, так что граф G содержит гамильтонов цикл.

Надеюсь, мне не нужно повторять хорошо знакомое завершение доказательства?

Наидлиннейший ациклический путь

В версии принятия решения ***задачи о наидлиннейшем ациклическом пути*** нам даны неориентированный граф G и целое число k и спрашивается, имеются ли в графе G две вершины, между которыми имеется ациклический путь не менее чем из k ребер.

И вновь, сертификат для данной задачи проверить очень легко. Он состоит из упорядоченного списка вершин в предлагаемом пути. Мы можем за полиномиальное время проверить, что список содержит по крайней мере $k+1$ вершину ($k+1$, потому что путь с k ребрами состоит из $k+1$ вершины), при этом ни одна вершина в пути не повторяется, и для каждой пары соседних вершин в списке имеется соединяющее их ребро.

Еще одно простое приведение показывает, что эта задача является NP-сложной. Мы выполняем приведение к ней задачи о гамильтоновом пути. Если в качестве входного для задачи о гамильтоновом пути дан граф G с n вершинами, то входными данными задачи о наидлиннейшем ациклическом пути являются тот же граф G и целое число $k=n-1$. Если это не приведение за полиномиальное время, то я уж и не знаю, что тогда можно назвать таковым...

Сумма подмножества

В ***задаче о сумме подмножества*** входными данными являются конечное неупорядоченное множество S положительных целых чисел и целевое число t, которое также является положительным целым числом. Мы хотим выяснить, существует ли подмножество S' множества S, сумма элементов которого в точности равна t. Например, если S представляет собой множество $\{1, 2, 7, 14, 49, 98, 343, 686, 2409, 2793, 16\,808, 17\,206, 117\,705, 117\,993\}$, а $t = 138\,457$, то решением является подмножество $S' = \{1, 2, 7, 98, 343, 686, 2409, 17\,206, 117\,705\}$. Сертификатом является, конечно же, подмножество S, проверить которое можно простым сложением чисел подмножества и сравнением суммы с числом t.

Как можно увидеть из генеалогического дерева NP-полноты на с. 185, демонстрировать NP-сложность задачи суммы подмножества мы будем путем приведения к ней задачи о 3-CNF-выполнимости. Это еще один пример приведения задачи из одной предметной области к задаче из другой, превращая логическую задачу в арифметическую. Это преобразование, несмотря на всю интеллектуальность, в конечном счете довольно простое.

Мы начинаем с булевой 3-CNF-формулы F, которая имеет n переменных и k выражений. Назовем переменные $v_1, v_2, v_3, \ldots, v_n$, а выражения — $C_1, C_2, C_3, \ldots, C_k$. Каждое выражение содержит ровно три литерала (вспомните, что каждый литерал представляет собой v_i или NOT v_i), объединенные операторами OR, а вся формула имеет вид $F = C_1$ AND C_2 AND C_3 AND $\cdots$ AND C_k. Таким образом, каждое выражение выполняется, если любой из его литералов равен единице, а полная формула F выполняется, только когда выполняются все входящие в нее выражения.

Прежде чем мы построим множество S для задачи суммы подмножества, сконструируем целевое число t из 3-CNF-формулы F. Будем строить его как десятичное число с $n+k$ цифрами. Младшие k цифр (k цифр справа) числа t соответствуют k выражениям формулы F, и каждая из этих цифр равна 4. Старшие n цифр числа t соответствуют n переменным формулы F, и каждая из этих цифр равна 1. Если формула F имеет, скажем, три переменные и четыре выражения, то t оказывается равным 1114444. Как мы увидим, если есть подмножество S, сумма которого равна t, то цифры t, которые соответствуют переменным (единицы), обеспечат присвоение значения каждой переменной в F, а цифры t, которые соответствуют выражениям (четверки) гарантируют выполнение каждого выражения F.

Множество S будет состоять из $2n+2k$ целых чисел. Оно содержит целые числа x_i и x'_i для каждой из n переменных v_i в 3-CNF-формуле F и целые числа q_j и q'_j для каждого из k выражений C_j в формуле F. Мы строим каждое целое число в S цифра за цифрой в десятичной системе счисления. Давайте рассмотрим пример с $n = 3$ переменными и $k = 4$ выражениями, так что 3-CNF-формула имеет вид $F = C_1$ AND C_2 AND C_3 AND C_4, и пусть выражения представляют собой

$$C_1 = v_1 \text{ OR } (\text{NOT } v_2) \text{ OR } (\text{NOT } v_3),$$
$$C_2 = (\text{NOT } v_1) \text{ OR } (\text{NOT } v_2) \text{ OR } (\text{NOT } v_3),$$
$$C_3 = (\text{NOT } v_1) \text{ OR } (\text{NOT } v_2) \text{ OR } v_3,$$
$$C_4 = v_1 \text{ OR } v_2 \text{ OR } v_3.$$

Вот как выглядят соответствующее множество S и целевое число t.

		v_1	v_2	v_3	C_1	C_2	C_3	C_4
x_1	=	1	0	0	1	0	0	1
x_1'	=	1	0	0	0	1	1	0
x_2	=	0	1	0	0	0	0	1
x_2'	=	0	1	0	1	1	1	0
x_3	=	0	0	1	0	0	1	1
x_3'	=	0	0	1	1	1	0	0
q_1	=	0	0	0	1	0	0	0
q_1'	=	0	0	0	2	0	0	0
q_2	=	0	0	0	0	1	0	0
q_2'	=	0	0	0	0	2	0	0
q_3	=	0	0	0	0	0	1	0
q_3'	=		0	0	0	0	2	0
q_4	=	0	0	0	0	0	0	1
q_4'	=	0	0	0	0	0	0	2
t	=	1	1	1	4	4	4	4

Обратите внимание, что заштрихованные элементы S — 1000110, 101110, 10011, 1000, 2000, 200, 10, 1 и 2 — дают в сумме 1114444. Вскоре мы увидим, чему эти элементы соответствуют в 3-CNF-формуле F.

Мы строим целые числа в S цифра за цифрой так, чтобы каждый столбец в приведенной таблице имел либо сумму 2 (n столбцов слева), либо 6 (k столбцов справа). Обратите внимание, что при суммировании элементов в S необходимости переноса не возникает ни в одной из позиций цифр, так что мы можем работать с числами цифра за цифрой.

Каждая строка в таблице помечена элементом из S. Первые $2n$ строк соответствуют n переменным в 3-CNF-формуле, а последние $2k$ строк представляют собой "слабину", цель которой мы увидим чуть позже. Строки, обозначенные элементами x_i и x_i', отвечают соответственно литералам v_i и NOT v_i в формуле F. Мы будем говорить, что строки "являются" литералами, понимая, что мы имеем в виду, что они соответствуют литералам. Наша цель заключается в том, чтобы включить в подмножество S' ровно n из $2n$ первых строк — ровно по одной строке из каждой пары x_i и x_i' — строки, которые будут соответствовать выполняющему набору 3-CNF-формулы F. Поскольку мы требуем, чтобы выбираемые строки добавляли по 1 в каждом из n левых столбцов, мы гарантируем, что

для каждой переменной v_i в 3-CNF-формуле в подмножество S' из двух строк для каждой пары x_i и x'_i будет включена только одна из них, но не обе одновременно. k столбцов справа гарантируют, что строки, которые мы включаем в S', представляют собой литералы, которые удовлетворяют каждое из выражений в 3-CNF-формуле.

Давайте ненадолго сосредоточимся на n левых столбцах, которые помечены переменными $v_1, v_2, \ldots, v_n$. Для каждой заданной переменной v_i в строках x_i и x'_i цифра, соответствующая v_i, равна единице, а цифры в позициях, соответствующих другим переменным, равны нулю. Например, три левые цифры для x_2 и x'_2 равны 010. Цифры в последних $2k$ строках в n левых столбцах нулевые. Поскольку целевое значение t содержит по единице в каждой позиции переменных, чтобы внести свой вклад в сумму, в подмножество S' для каждого i должно входить ровно одно значение — либо x_i, либо x'_i (они не могут ни одновременно входить в S', ни одновременно отсутствовать в этом подмножестве). Наличие x_i в S' соответствует установке $v_i = 1$, а наличие x'_i соответствует $v_i = 0$.

Теперь обратим свое внимание на k правых столбцов, которые соответствуют выражениям. Эти столбцы, как мы увидим ниже, гарантируют, что каждое выражение выполняется. Если литерал v_i встречается в выражении C_j, то в строке x_i в столбце для C_j находится значение 1; если же в выражении C_j имеется литерал NOT v_i, то значение 1 в столбце C_j находится в строке x'_i. Поскольку каждое выражение в 3-CNF-формуле содержит ровно три разных литерала, столбец каждого выражения должен содержать ровно три единицы среди всех строк x_i и x'_i. Для данного выражения C_j среди первых $2n$ строк строки, включенные в S', соответствуют выполнению 0, 1, 2 или 3 литералов в C_j, так что эти строки добавляют к общей сумме столбца C_j значения 0, 1, 2 или 3.

Однако целевой цифрой для каждого выражения является 4, и вот тут и вступают в игру "слабины" q_j и q'_j для $j = 1, 2, 3, \ldots, k$. Они гарантируют, что для каждого выражения подмножество S' включает некоторый литерал из этого выражения (некоторое x_i или x'_i, которое имеет значение 1 в соответствующем столбце). Строка для q_j имеет единицу в столбце для выражения C_j и нуль — во всех остальных; строка для q'_j такая же, за исключением того, что вместо значения 1 она содержит значение 2. Мы можем суммировать эти строки для достижения целевой цифры 4, но только если подмножество S' включает по крайней мере один литерал из C_j. Какие из этих строк "слабин" должны быть суммированы, зависит от количества литералов выражения C_j, включенных в S'. Если S' включает только один литерал, то необходимы обе строки, поскольку сумма в столбце состоит из единицы от литерала, плюс единица от q_j и двойка от q'_j. Если S' включает два литерала, то нужна лишь строка q'_j, так как два литерала дают в сумме двойку. Если же S' включает три литерала, то нужна строка q_j, поскольку три литерала вносят в сумму вклад, равный 3, так что для получения значения 4 требуется одна единица — от строки q_j. Но если в S' не входит ни один литерал из C_j, то $q_j + q'_j = 3$ недостаточно для получения целевого значения 4. Следовательно, достичь целевого значения 4 можно, только если в подмножество S' входит некоторый литерал из выражения.

Теперь, когда мы познакомились с приведением, убедимся, что оно выполнимо за полиномиальное время. Мы создаем $2n + 2k + 1$ целых чисел (включая целевое значение t),

каждое из которых состоит из $n+k$ цифр. Из диаграммы видно, что все создаваемые целые числа различны, так что S действительно представляет собой множество (определение множества не допускает наличия повторяющихся элементов).

Чтобы показать, что данное приведение работает, нужно показать, что 3-CNF-формула F имеет выполняющий набор тогда и только тогда, когда существует подмножество S' множества S, сумма элементов которого в точности равна t. Вы уже встречались с этой идеей, но давайте повторимся. Во-первых, предположим, что F имеет выполняющий набор. Если в этом наборе $v_i = 1$, включим x_i в S'; в противном случае включим x_i'. Поскольку в S входит только один из элементов x_i и x_i', столбец для v_i должен давать в сумме значение 1, равное соответствующей цифре числа t. Поскольку при наличии выполняющего набора выполняется каждое выражение C_j, строки x_i и x_i' должны давать в сумму в столбце C_j значение 1, 2 или 3 (количество литералов C_j, равных 1). Включение необходимых строк слабины q_j и/или q_j' в S' дает требуемую целевого цифру 4.

И обратно, предположим, что S имеет подмножество S', сумма элементов которого в точности равна t. Чтобы число t имело единицы в n левых позициях, S' должно включать ровно один из элементов x_i и x_i' для каждой переменной v_i. Если оно включает x_i, устанавливаем $v_i = 1$; если же оно включает x_i', устанавливаем $v_i = 0$. Поскольку строки слабины q_j и q_j', просуммированные вместе, не могут дать целевую цифру 4 в столбце для выражения C_j, подмножество S' должно включать по крайней мере одну из строк x_i или x_i' с единицей в столбце C_j. Если оно включает x_i, то в выражении C_j имеется литерал v_i, и выражение выполняется. Если S' включает x_i', то в выражении C_j имеется литерал NOT v_i, и выражение, опять же, выполняется. Таким образом, выполняется каждое из выражений, а значит, для 3-CNF-формулы F имеется выполняющий набор.

Таким образом, мы видим, что если можно за полиномиальное время решить задачу о сумме подмножества, то тогда за полиномиальное время можно определить, является ли 3-CNF-формула выполнимой. Поскольку задача о 3-CNF-выполнимости является NP-полной, таковой же является и задача о сумме подмножества. Кроме того, если мы знаем, какие целые числа множества S при суммировании дают целевое значение t, то мы можем определить и выполняющий набор переменных для 3-CNF-формулы.

Еще одно замечание о рассмотренном приведении: цифры не обязаны быть десятичными. Важно, чтобы при суммировании не происходили переносы из разряда в разряд, а поскольку сумма столбца не может превышать 6, можно использовать любую систему счисления, лишь бы ее основание было не менее 7.

Разбиение

Задача о разбиении тесно связана с задачей о сумме подмножества. Фактически это частный случай последней: если z равно сумме всех целых чисел в множестве S, то целевое значение t в точности равно $z/2$. Другими словами, цель заключается в том, чтобы определить, существует ли разбиение множества S на два непересекающихся множества S' и S'' таким образом, чтобы каждое целое число из S находилось либо в S', либо в S'', но не в обоих подмножествах одновременно (что, собственно, и означает термин "разбие-

ние S на S' и S''"), и чтобы сумма чисел в S' была равна сумме чисел в S''. Как и в задаче о сумме подмножества, сертификат представляет собой подмножество множества S.

Чтобы показать, что задача о разбиении является NP-сложной, приведем к ней задачу о сумме подмножества (вряд ли это решение показалось вам удивительным). Для входных данных задачи, а именно — для заданного множества R положительных целых чисел и целевого положительного целого числа t, мы строим за полиномиальное время множество S, являющееся входными данными к задаче разбиения. Во-первых, вычислим сумму z всех целых чисел в R. Мы предполагаем, что $z \neq 2t$, потому что если это так, то задача уже представляет собой задачу разбиения. (Если $z = 2t$, то $t = z/2$, и мы пытаемся найти подмножество R, сумма элементов которого та же, что и сумма элементов, не входящих в него.) Затем выберем любое целое число y, большее, чем $t+z$ и $2z$. Определим множество S как содержащее все целые числа из R и два дополнительных целых числа $y-t$ и $y-z+t$. Поскольку y больше, чем $t+z$ и $2z$, мы знаем, что и $y-t$, и $y-z+t$ больше z (суммы всех чисел в R), так что таких чисел в R быть не может. (Вспомните, что поскольку S представляет собой множество, все его элементы должны быть различными. Мы также знаем, что, поскольку $z \neq 2t$, должно выполняться неравенство $y-t \neq y-z+t$, так что два новых целых числа также уникальны.) Заметим, что сумма всех целых чисел в S равна $z+(y-t)+(y-z+t)$, т.е. просто $2y$. Таким образом, если S разделить на два непересекающихся подмножества с равными суммами, сумма каждого из подмножеств должна быть равна y.

Чтобы показать, что описанное приведение работает, нужно показать, что существует подмножество R' множества R, целые числа которого дают в сумме t тогда и только тогда, когда существует разбиение множества S на подмножества S' и S'', такое, что сумма чисел в S' совпадает с суммой чисел в S''. Во-первых, давайте предположим, что некоторое подмножество R' множества R имеет целые числа, сумма которых равна t. Тогда сумма целых чисел из R, которые не входят в R', должна быть равна $z-t$. Определим множество S' как содержащее все целые числа из R' вместе с $y-t$ (так что S'' содержит $y-z+t$ вместе со всеми целыми числами из R, которые не входят в R'). Нам просто нужно показать, что целые числа в S' в сумме дают y. Но это легко: сумма целых чисел в R' равна t, так что добавление $y-t$ дает сумму y.

И обратно, давайте предположим, что существует разбиение множества S на S' и S'', суммы элементов которых равны y. Я утверждаю, что два целых числа, которые мы добавили к множеству R при построении множества S ($y-t$ и $y-z+t$) не могут одновременно находиться в одном подмножестве S' или S''. Почему? Если бы они оба были в одном подмножестве, то сумма элементов этого подмножества была бы не менее чем $(y-t)+(y-z+t)=2y-z$. Но вспомним, что y больше z (на самом деле y больше $2z$), а потому $2y-z$ больше, чем y. Таким образом, если числа $y-t$ и $y-z+t$ находятся в одном подмножестве, его сумма будет больше, чем y. Поэтому мы точно знаем, одно из чисел $y-t$ и $y-z+t$ находится в S', а второе — в S''. Не имеет значения, в каком именно подмножестве находится $y-t$, так что пусть это будет подмножество S'. Мы знаем, что сумма целых чисел, входящих в S', равна y, а это означает, что сумма целых чисел в S', отличных от числа $y-t$, равна $y-(y-t)=t$. Поскольку число $y-z+t$ находиться в S' не может, все

числа в S', кроме $y-t$, входят в множество R. Следовательно, существует подмножество R, сумма элементов которого равна t.

Рюкзак

В ***задаче о рюкзаке*** задано множество из n элементов, каждый из которых имеет свой вес и цену, и требуется указать, существует ли такое подмножество элементов, что их общий вес не превышает W, а их суммарная стоимость при этом не меньше V. Эта задача представляет собой версию принятия решения задачи оптимизации, в которой требуется загрузить рюкзак самым ценным подмножеством элементов, не превышающим предельный вес. Эта задача оптимизации имеет очевидные приложения, такие как решение о том, что следует взять с собой в поход или что должен выносить из обворованного дома грабитель.

Задача разбиения представляет собой частный случай задачи о рюкзаке, в котором цена каждого элемента равна его весу, а W и V равны половине общего веса. Если бы мы могли решить задачу о рюкзаке за полиномиальное время, то могли бы решить за полиномиальное время и задачу разбиения множества. Таким образом, задача о рюкзаке как минимум так же сложна, как и задача о разбиении множества, и нам даже не нужно проходить через полный процесс приведения, чтобы показать NP-полноту задачи о рюкзаке.

Общие стратегии

Как вы уже, вероятно, поняли к настоящему времени, не существует универсального способа приведения одной задачи к другой с целью доказательства NP-сложности. Некоторые приведения довольно просты, как приведение задачи о гамильтоновом цикле к задаче коммивояжера, а некоторые чрезвычайно сложны. Вот несколько принципов и стратегий, о которых следует помнить, так как они часто оказываются полезными.

Идти от общего к частному

При приведении задачи X к задаче Y вы всегда должны начинать с произвольных входных данных для задачи X. Но входные данные задачи Y вы можете ограничить так, как вам нравится. Например, при приведении задачи 3-CNF-выполнимости к задаче о сумме подмножества приведение в состоянии обработать *любую* 3-CNF-формулу в качестве входных данных, но сгенерированные приведением входные данные для задачи о сумме подмножества имеют определенную структуру, а именно — $2k+2n$ целых чисел в множестве, причем каждое целое число сформировано определенным образом. Приведение не в состоянии сгенерировать *каждые* возможные входные данные для задачи о сумме подмножества, но это и не нужно. Дело в том, что мы можем решить задачу 3-CNF-выполнимости путем преобразования входных данных во входные данные для задачи о сумме подмножества, а затем с помощью ответа на последнюю получить ответ и для задачи 3-CNF-выполнимости.

Заметьте, однако, что каждое приведение должно иметь один и тот же вид: оно должно преобразовывать *любые* входные данные задачи X в *некоторые* входные данные задачи

Y даже при объединении приведений в цепочку. Если вы хотите привести задачу *X* к задаче *Y*, а задачу *Y* к задаче *Z*, то первое приведение должно превратить *любые* входные данные задачи *X* в *некоторые* входные данные *Y*, а второе приведение должно превратить *любые* входные данные задачи *Y* в *некоторые* входные данные задачи *Z*. Если второе приведение преобразует во входные данные для задачи *Z* только те входные данные, которые генерируются первым приведением для задачи *Y* из входных данных для задачи *X*, этого недостаточно.

Использовать преимущества ограничений приводимой задачи

В общем случае при приведении задачи *X* к задаче *Y* можно выбрать задачу *X* так, чтобы наложить дополнительные ограничения на входные данные. Например, почти всегда гораздо проще выполнить приведение задачи о 3-CNF-выполнимости, чем приводить первичную задачу выполнимости булевых формул общего вида. Булевы формулы могут быть произвольной сложности, но вы уже видели, как можно использовать при приведении структуру 3-CNF-формул.

Кроме того, обычно более просто выполнять приведение задачи о гамильтоновом цикле, чем приведение задачи о коммивояжере, даже несмотря на их сильную схожесть. Дело в том, что в задаче коммивояжера веса ребер могут быть любыми положительными числами, а не только нулем или единицей, как мы требуем при приведении к данной задаче. Задача о гамильтоновом цикле оказывается более ограниченной, потому что каждое ребро может иметь только одно из двух "значений": присутствует или отсутствует.

Рассмотрение частных случаев

Несколько NP-полных задач представляют собой всего лишь частные случаи других NP-полных задач, как, например, задача разбиения является частным случаем задачи о рюкзаке. Если вы знаете, что задача *X* является NP-полной и что это частный случай задачи *Y*, то задача *Y* также должна быть NP-полной. Дело в том, что, как мы видели в случае задачи о рюкзаке, полиномиальное время решения задачи *Y* автоматически даст решение задачи *X* за полиномиальное время. Интуитивно задача *Y*, будучи более общей, чем задача *X*, не может быть проще последней.

Выбор подходящей задачи для приведения

Зачастую верной для доказательства NP-полноты оказывается стратегия выбора приводимой задачи из той же, или по крайней мере тесно связанной, предметной области. Например, мы показали NP-полноту задачи вершинного покрытия (задачи, связанной с графами) путем приведения к ней задачи о клике, также связанной с графами. Генеалогическое дерево NP-полных задач показывает, что далее выполняется приведение к задачам о гамильтоновом цикле, гамильтоновом пути, о коммивояжере и о наидлиннейшем ациклическом пути — все они представляют собой задачи на графах.

Однако иногда оказывается оправданным переход от одной предметной области к другой, например когда мы приводили задачу о 3-CNF-выполнимости к задаче о клике или

к задаче о сумме подмножества. Когда приходится переходить от одной предметной области к другой, задача о 3-CNF-выполнимости часто оказывается хорошим выбором.

Когда в задачах на графах требуется выбрать часть графа без учета упорядоченности, хорошим стартом может оказаться задача о вершинном покрытии. Если важную роль играет упорядочение, рассмотрите возможность воспользоваться задачами о гамильтоновом цикле или гамильтоновом пути.

Увеличение мотивации

При преобразовании входного графа G задачи о гамильтоновом цикле во взвешенный граф G', выступающий в качестве входных данных для задачи коммивояжера, мы в действительности хотели бы, чтобы при решении задачи коммивояжера выбор ребер осуществлялся среди тех ребер, которые имеются в графе G. Чтобы мотивировать такой выбор, мы придали этим ребрам очень низкий нулевой вес. Иными словами, использование этих ребер мы сделали выгодным.

Мы могли бы также дать ребрам из графа G некоторый конечный вес, а ребрам, в G отсутствующим, — бесконечный, тем самым назначая "штраф" за использование ребер, отсутствующих в G. Если бы мы приняли этот подход и назначили каждому ребру из G вес W, то нам бы пришлось установить целевой вес k всего тура коммивояжера равным nW.

Разработка виджетов

Я не углубляюсь в эту тему, потому что виджеты могут быть очень сложными. Они могут оказаться полезными для того, чтобы принудительного обеспечить определенные свойства при построении приведения. Книги, приведенные в разделе "Дальнейшее чтение", содержат множество примеров построения и использования виджетов в приведениях.

Перспективы

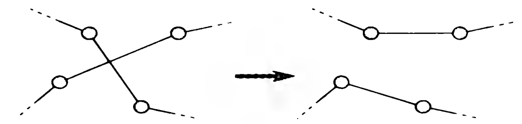
Я изобразил здесь довольно мрачную картину, не так ли? Представьте себе, как вы тратите время, силы, нервы, пытаясь найти алгоритм с полиномиальным временем для решения проблемы, но независимо от приложенных усилий, вы просто ничего не можете сделать. Через некоторое время вы были бы счастливы, если бы могли найти алгоритм со временем $O(n^5)$, несмотря на то что n^5 — это ужасно быстрый рост. Может быть, эта задача очень похожа на задачу, для которой вы знаете решение за полиномиальное время (вспомните задачи о 2-CNF- и 3-CNF-выполнимости или эйлеров и гамильтонов циклы), и вас ужасно расстраивает, что никак не получается адаптировать известный алгоритм с полиномиальным временем работы для вашей задачи. В конце концов вы начинаете подозревать, что, может быть — только может быть! — вы просто бьетесь головой о стену, пытаясь решить NP-полную задачу? И вот, наконец, у вас получается привести к вашей задаче какую-то из известных NP-полных задач, и теперь вы точно знаете, что ваша задача NP-сложная.

Все, это конец? Нет никакой надежды, что вы сможете решить задачу за какое-то разумное время?

Не совсем. Если задача NP-полная, это означает, что для некоторых входных данных решать задачу очень трудно, но ведь это не значит, что плохими будут *все* входные данные. Например, поиск наидлиннейшего ациклического пути в ориентированном графе является NP-полной задачей, но если известно, что входной граф ацикличен, то найти наидлиннейший ациклический путь можно не просто за полиномиальное время, а за время $O(n+m)$ (если граф имеет n вершин и m ребер). Напомним, что мы решили именно эту задачу, найдя критический путь в диаграмме PERT в главе 5, "Ориентированные ациклические графы". Еще один пример — если вы пытаетесь решить задачу разбиения множества целых чисел, и при этом сумма всех чисел нечетна, то вы сразу можете сказать, что разбить это множество на два с одинаковыми суммами элементов невозможно.

Хорошие новости выходят за рамки таких патологических частных случаев. Далее мы сосредоточимся на задачах оптимизации, версии принятия решения которых являются NP-полными задачами, таких как задача коммивояжера. Некоторые быстрые методы дают хорошие, а часто и очень хорошие результаты. Метод ***ветвления с отсечением*** организует поиск оптимального решения в древовидной структуре и отсекает куски этого дерева, убирая тем самым большие части пространства поиска. Метод основан на той простой идее, что если можно определить, что все решения, исходящие из некоторого узла дерева поиска, не могут оказаться лучше, чем наилучшее решение, найденное к настоящему моменту, то этот узел со всеми исходящими из него ветвями можно просто отсечь.

Другой часто срабатывающий метод — поиск в окрестности, который получает некоторое решение и пытается улучшить его с помощью некоторых локальных операций, до тех пор, пока это возможно. Рассмотрим задачу коммивояжера, в которой все вершины представляют собой точки на плоскости, а вес каждого ребра равен расстоянию между соответствующими точками на плоскости. Даже с учетом этого ограничения задача остается NP-полной. Один из методов оптимизации решения состоит в том, чтобы всякий раз, когда два ребра пересекаются, выполнять показанное ниже "переключение" путей, приводящее к более короткому циклу.



Кроме того, множество ***приближенных алгоритмов*** дают результаты, которые будут отличаться от оптимального решения не более чем на гарантированный множитель. Например, если входные данные для задачи коммивояжера подчиняются неравенству треугольника (для всех вершин u, v и x вес ребра (u,v) не превышает суммы весов ребер (u,x) и (x,v)), то имеется простой приближенный алгоритм, который всегда находит маршрут, общий вес которого не более чем в два раза превышает вес оптимального маршрута, и при этом время его работы линейно зависит от размера входных данных. Существует еще лучший приближенный алгоритм с полиномиальным временем работы для описанной ситуации, дающий маршрут, общий вес которого отличается от веса оптимального маршрута не более чем в $3/2$ раза.

Это может показаться странным, но если две NP-полные задачи достаточно тесно связаны между собой, то решения, получаемые хорошим приближенным алгоритмом для одной из них, для другой могут оказаться никуда не годными. То есть решение, которое оказывается близким к оптимальному для одной из задач, необязательно отображается на решение, сколь-нибудь близкое к оптимальному для другой задачи.

Тем не менее во многих реальных ситуациях близкое к оптимальному решение оказывается достаточно хорошим. Возвращаясь к примеру с компанией по доставке товаров, можно утверждать, что руководство этой компании будет радо найти почти оптимальные маршруты для своих грузовиков, даже если эти маршруты и не будут наилучшими. Ценен каждый доллар, который они смогут сэкономить путем планирования маршрутов.

Неразрешимые задачи

Если вы решили, что NP-полные задачи — самые трудные в мире алгоритмов, вас ждет очередное разочарование. Ученые-информатики определили большую иерархию классов сложности, основанную на том, сколько времени и иных ресурсов необходимо для решения задачи. Некоторые задачи требуют для решения времени, доказуемо экспоненциально зависящего от размера входных данных.

Но бывает и еще хуже. Для некоторых задач алгоритм решения существовать не может. То есть имеются задачи, для которых доказательно невозможно создать алгоритм, который всегда дает правильный ответ. Мы называем такие задачи ***неразрешимыми***, и наиболее известна среди них ***задача останова***, неразрешимость которой доказана математиком Аланом Тьюрингом (Alan Turing) в 1937 году. Входными данными задачи останова является компьютерная программа A и ее входные данные x. Цель заключается в том, чтобы определить, остановится ли когда-либо программа A при работе с входными данными x. Другими словами, завершится ли когда-либо обработка программой A входных данных x?

Возможно, вы думаете, что могли бы написать программу — назовем ее программой B, — которая считывает программу A, ее входные данные x и имитирует работу программы A с данными x. Да, это работает, если программа A при работе с входными данными x в конечном итоге завершается. Но что делать, если это не так? Как программа B может узнать, что следует объявить, что программа A никогда не остановится? Может ли программа B выяснить, не попала ли программа A в своего рода бесконечный цикл? Ответ таков — хотя и можно написать программу B для проверки некоторых случаев, когда программа A не завершается, доказуемо невозможно написать программу B так, чтобы *она* всегда останавливалась и верно сообщала о том, завершится ли программа A при входных данных x.

Поскольку невозможно написать программу, которая определяет, завершится ли другая программа при работе с конкретными входными данными, невозможно также написать программу, которая определяет, соответствует ли другая программа своей спецификации. Как может программа сказать, даст ли другая программа верный ответ, если она не в состоянии даже сказать, завершит ли та свою работу? Так что идеальное автоматизированное тестирование программного обеспечения, как видим, невозможно.

Чтобы вы не думали, что неразрешимые задачи связаны только со свойствами компьютерных программ, ***задача соответствия Поста*** (Post correspondence problem) связана со строками, с которыми мы встречались в главе 7, "Алгоритмы на строках". Предположим, что у нас есть по крайней мере два символа, и два списка из n строк, A и B, состоящих из этих символов. Пусть список A состоит из строк $A_1, A_2, A_3, \ldots, A_n$, а список B — из строк $B_1, B_2, B_3, \ldots, B_n$. Задача заключается в выяснении, существует ли последовательность индексов $i_1, i_2, i_3, \ldots, i_m$, такая, что $A_{i_1} A_{i_2} A_{i_3} \cdots A_{i_m}$ (то есть конкатенация строк $A_{i_1}, A_{i_2}, A_{i_3}, \ldots, A_{i_m}$) дает ту же строку, что и $B_{i_1} B_{i_2} B_{i_3} \cdots B_{i_m}$. Пусть, например, в роли символов выступают `e`, `h`, `m`, `n`, `o`, `r` и `y`, что $n = 5$ и что

$$
\begin{aligned}
A_1 &= \texttt{ey}, & B_1 &= \texttt{ym}, \\
A_2 &= \texttt{er}, & B_2 &= \texttt{r}, \\
A_3 &= \texttt{mo}, & B_3 &= \texttt{oon}, \\
A_4 &= \texttt{on}, & B_4 &= \texttt{e}, \\
A_5 &= \texttt{h}, & B_5 &= \texttt{hon}.
\end{aligned}
$$

Тогда одним из решений является последовательность индексов $\langle 5,4,1,3,4,2 \rangle$, поскольку и $A_5 A_4 A_1 A_3 A_4 A_2$, и $B_5 B_4 B_1 B_3 B_4 B_2$ образуют строку `honeymooner`. Конечно, если есть одно решение, есть и бесконечное их количество, поскольку вы можете просто повторять последовательность индексов (получая `honeymoonerhoneymooner` и т.д.). Чтобы задача соответствия Поста была неразрешимой, мы должны позволить использовать строки из A и B несколько раз, так как в противном случае можно было бы просто перечислить все возможные комбинации строк.

Хотя задача соответствия Поста может показаться не особенно интересной сама по себе, ее можно привести к другим задачам, чтобы показать, что они тоже алгоритмически неразрешимы. Это та же основная идея, которую мы использовали, чтобы показать, что задача является NP-сложной: заданный экземпляр задачи соответствия Поста приводится к экземпляру другой задачи Q, так что ответ на экземпляр задачи Q дает ответ на экземпляр задачи соответствия Поста. Если мы могли бы решить задачу Q, то тем самым мы могли бы решить и задачу соответствия Поста. Но поскольку мы знаем, что задача соответствия Поста неразрешима, то неразрешимой должна быть и задача Q.

Среди неразрешимых задач, к которым мы можем привести задачу соответствия Поста, имеются несколько задач, связанных с ***контекстно-свободными грамматиками***, которые описывают синтаксис большинства языков программирования. Контекстно-свободная грамматика — это набор правил генерации ***формального языка***, который представляет собой причудливый способ вывода "множества строк". Выполняя приведение задачи соответствия Поста, можно доказать, что неразрешимыми являются такие задачи, как выяснение, генерируют ли две контекстно-свободные грамматики один и тот же формальный язык или является ли данная контекстно-свободная ***неоднозначной***, т.е. существуют ли два различных способа генерации одной и той же строки с помощью правил данной контекстно-свободной грамматики.

Итоги

Мы встретили в книге широкий спектр алгоритмов из самых разных предметных областей. Мы видели алгоритм с сублинейным временем работы — бинарный поиск. Мы видели алгоритмы с линейным временем работы — линейный поиск, сортировка подсчетом, поразрядная сортировка, топологическая сортировка и поиск кратчайшего пути в ориентированном ациклическом графе. Мы видели алгоритмы со временем работы $O(n \lg n)$ — сортировка слиянием и быстрая сортировка (средний случай). Мы видели алгоритмы со временем работы $O(n^2)$ — сортировка выбором, сортировка вставкой и быстрая сортировка (в наихудшем случае). Мы видели алгоритмы на графах, время работы которых описывается некоторой нелинейной комбинацией числа n вершин и числа m ребер, — алгоритм Дейкстры и алгоритм Беллмана–Форда. Мы видели алгоритм на графе со временем работы $O(n^3)$ — алгоритм Флойда–Уоршелла. Наконец, теперь мы знаем, что для некоторых задач неизвестно, имеется ли алгоритм их решения за полиномиальное время работы. Более того, мы знаем, что для некоторых задач алгоритм решения просто не существует, независимо от времени его работы.

Даже из этого относительно краткого введения в мир компьютерных алгоритмов[5] видно, насколько широк охват этой предметной области. Эта книга охватывает лишь некоторые небольшие островки в безбрежном океане. Кроме того, я ограничил наш анализ конкретной моделью вычислений, в которой операции выполняет только один процессор, а время выполнения каждой операции является более или менее одинаковым независимо от того, где в памяти компьютера располагаются данные. Имеется множество альтернативных вычислительных моделей — были предложены модели с несколькими процессорами; модели, в которых время выполнения операции зависит от расположения данных; модели, в которых данные поступают в виде однонаправленного неповторяющегося потока; модели, в которых компьютер представляет собой квантовое устройство.

Главное — вы теперь видите, что на карте этого безбрежного океана еще много белых пятен, много вопросов без ответа, много вопросов, которые еще даже не заданы. Так что у вас есть отличная возможность проявить себя, выбрав эту область знаний для изучения и работы! Дерзайте!

Дальнейшее чтение

В первую очередь, это книга об NP-полноте Гари (Garey) и Джонсона (Johnson) [7]. Если вас интересует эта тема — прочтите ее. В CLRS [4] имеется посвященная NP-полноте глава, в которой гораздо больше технических деталей, чем в этой книге; кроме того, в ней также есть глава о приближенных алгоритмах. Более подробно о вычислимости и сложности, а также (кратко и понятно) о задаче останова можно прочесть в книге Сипсера (Sipser) [19].

[5] *Сравните эту книгу с CLRS — более 1300 страниц в третьем издании.*

Библиография

1. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
2. Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin, and Robert E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37(2):213–223, 1990.
3. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, first edition, 1990.
4. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.
   Имеется перевод на русский язык: Кормен Т., Лейзесон Ч., Ривест Р., Штайн К. *Алгоритмы: построение и анализ, 3-е издание*. — М.: ООО "И.Д. Вильямс", 2013.
5. Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.
6. Annex C: Approved random number generators for FIPS PUB 140-2, Security requirements for cryptographic modules. http://csrc.nist.gov/ publications/fips/fips140-2/ fips1402annexc.pdf, July 2011. Draft.
7. Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
8. David Gries. *The Science of Programming*. Springer, 1981.
9. Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2008.
10. Donald E. Knuth. *The Art of Computer Programming*, Volume 1: Fundamental Algorithms. Addison-Wesley, third edition, 1997.
    Имеется перевод на русский язык: Кнут Д. *Искусство программирования, т. 1. Основные алгоритмы, 3-е изд.* — М.: ООО "И.Д. Вильямс", 2000.
11. Donald E. Knuth. *The Art of Computer Programming*, Volume 2: Seminumeral Algorithms. Addison-Wesley, third edition, 1998.
    Имеется перевод на русский язык: Кнут Д. *Искусство программирования, т. 2. Получисленные алгоритмы, 3-е изд.* — М.: ООО "И.Д. Вильямс", 2000.
12. Donald E. Knuth. *The Art of Computer Programming*, Volume 3: Sorting and Searching. Addison-Wesley, second edition, 1998.
    Имеется перевод на русский язык: Кнут Д. *Искусство программирования, т. 3. Сортировка и поиск, 2-е изд.* — М.: ООО "И.Д. Вильямс", 2000.
13. Donald E. Knuth. *The Art of Computer Programming*, Volume 4A: Combinatorial Algorithms, Part I. Addison-Wesley, 2011.
    Имеется перевод на русский язык: Кнут Д. *Искусство программирования, т. 4,А. Комбинаторные алгоритмы, часть 1.* — М.: ООО "И.Д. Вильямс", 2013.

14. John MacCormick. *Nine Algorithms That Changed the Future: The Ingenious Ideas That Drive Today's Computers*. Princeton University Press, 2012.

15. John C. Mitchell. *Foundations for Programming Languages*. TheMIT Press, 1996.

16. Alfred Menezes, Paul van Oorschot, and Scott Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

17. Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. See also U.S. Patent 4,405,829.

18. David Salomon. *A Concise Introduction to Data Compression*. Springer, 2008.

19. Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, second edition, 2006.

20. Sean Smith and John Marchesini. *The Craft of System Security*. Addison-Wesley, 2008.

21. James A. Storer. *Data Compression: Methods and Theory*. Computer Science Press, 1988.

22. Greg Taylor and George Cox. Digital randomness. *IEEE Spectrum*, 48(9):32–58, 2011.

Предметный указатель